A 3D DISPLAY SYSTEM FOR LIGHTNING DETECTION AND RANGING (LDAR) DATA

THESIS

Michael W. Darwin, 1st Lt, USAF

AFIT/GM/ENP/00M-05

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GM/ENP/00M-05

A 3-D DISPLAY SYSTEM FOR LIGHTNING DETECTION AND RANGING (LDAR) DATA

THESIS

Presented to the Faculty

Department of Engineering Physics

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Meteorology

Michael W. Darwin, B.S.

First Lieutenant, USAF

March, 2000

AFIT/GM/ENP/00M-05

A 3-D DISPLAY SYSTEM FOR LIGHTNING DETECTION AND RANGING (LDAR) DATA

Michael W. Darwin, B.S.
First Lieutenant, USAF

Approved:

Cecilia A. Miner (Chairman)	Date
Timothy M. Jacobs (Member)	Date
Gary R. Huffines (Member)	Date
David E. Weeks (Member)	Date

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense of the U. S. Government.

Acknowledgements

I would like to thank everyone who had a hand in helping me successfully complete my degree and thesis, especially the members of my committee, chaired by Lt Col Cecilia Miner.

I would also like to thank my thesis sponsor, Mr. William Roeder, Chief Staff Meteorologist of the 45th Weather Squadron at Patrick AFB and Cape Canaveral Air Station, Florida, for providing me with a topic for which I was able to actually do considerable computer graphics work.

Thanks also goes to the United States Air Force, for picking up the tab for my Bachelors and Masters degrees, and for giving me the opportunity to succeed.

Finally, I must thank my wife, Shawna, my daughter Caitlin, and my son Evan, for putting up with Dad's endless hours at school and slaving away on the computer.

Michael W. Darwin

Table of Contents

List of Figures

Abstract

Lightning detection is an essential part of safety and resource protection at Cape Canaveral. In order to meet the unique needs of launching space vehicles in the thunderstorm prone Florida environment, Cape Canaveral has the only operational three-dimensional (3D) lightning detection network in the world, the Lightning Detection and Ranging (LDAR) system. Although lightning activity is detected in three dimensions, the current LDAR display, developed 20 years ago, is two-dimensional. This thesis uses modern three-dimensional graphics, object-oriented software design, and innovative visualization techniques to develop a 3D visualization application for LDAR data.

The individual data points in an LDAR data file are compiled into a tree-like hierarchy using Java data structures. This hierarchy groups the points into a series of nested 3D cubes of varying sizes. The resulting data structures are used to construct a Java 3D scene graph containing the lightning information, using a visualization technique called Nested Cubes. Nested Cubes divides the Cape Canaveral area into a series of non-overlapping cubes 10 km on a side. If any stepped leaders are detected within one of these areas, they become visible in the scene as a transparent, red 10 km cube. If the user zooms in close enough, a 10 km cube will disappear and be replaced first by 1 km cubes, then 100 m cubes, bounding the areas where lightning was detected inside the larger cube. Finally, 100 meter cubes are replaced by individual points of 1 meter resolution, which are the actual points found in the original LDAR data file.

A 3D DISPLAY SYSTEM FOR LIGHTNING DETECTION AND RANGING (LDAR) DATA

1. Introduction

1.1 Overview

Cape Canaveral Air Station and the Kennedy Space Center (CCAS/KSC), located on the Atlantic side of the Florida coast, are important installations for the launch of United States (US) space vehicles. Weather has a large impact on CCAS/KSC space operations, in part because the Cape Canaveral area is one of the most lightning prone locations in the US, experiencing an average of 81 days with thunderstorms each year. Nearly 35 percent of all space launch attempts at CCAS/KSC are scrubbed or delayed due to weather (Roeder, 1999) in an average year. Lightning has an even larger impact on space launch by interfering with ground operations in preparation for launch. These ground operations are outside, exposing the workers to weather hazards such as lightning. CCAS/KSC averages over 5,000 ground operations each year, many of which must be postponed or cancelled due to an average of over 1200 lightning watches/warnings.

The 45th Weather Squadron (45 WS), based at Patrick Air Force Base, is responsible for weather forecasting for launch operations. In the interest of improving their support for the space launch mission, they sponsor research such as this thesis. Because of the large impact of lightning upon their operations, much of their research centers on lightning and thunderstorms, and how to lessen the impact of these phenomena

on their operations. In order to meet the needs of launching space vehicles in the difficult environment of Cape Canaveral, CCAS/KSC has a unique 3D lightning detection and display system known as Lightning Detection and Ranging (LDAR), developed in the late 1970s by the National Aeronautics and Space Administration (NASA). At that time, outfitting LDAR with a true 3D display would have been prohibitively expensive (Gallagher, 1995). Advances in computer graphics and computing power over the last 20 years, however, now make it possible to run a 3D display for LDAR from an ordinary PC or workstation. The purpose of this thesis is to create a 3D LDAR display using off-the-shelf hardware and modern object-oriented software design.

1.2 Background

The LDAR system was developed by NASA specifically for use at the Kennedy Space Center/Cape Canaveral Air Station. It is the only operational three-dimensional lightning detection network in the United States. Vandenberg AFB, CA is the other major space port in the US, but does not have an LDAR system since they are in the area of lowest lightning activity in the US. Two-dimensional lightning location is sufficient for airfields because planes can change their direction of flight once they take off, but a space vehicle follows a predetermined flight path as it rises through the atmosphere, and before it can be launched this three-dimensional path must be clear of lightning and the potential for lightning. Therefore the CCAS/KSC area is the only location with a permanent three-dimensional lightning detection system in place, and is the only regular operational weather source of 3D lightning data in the US.

The current LDAR display does not easily lend itself to 3D visualization. It consists of three two-dimensional display areas, one each for the XY, XZ, and YZ planes.

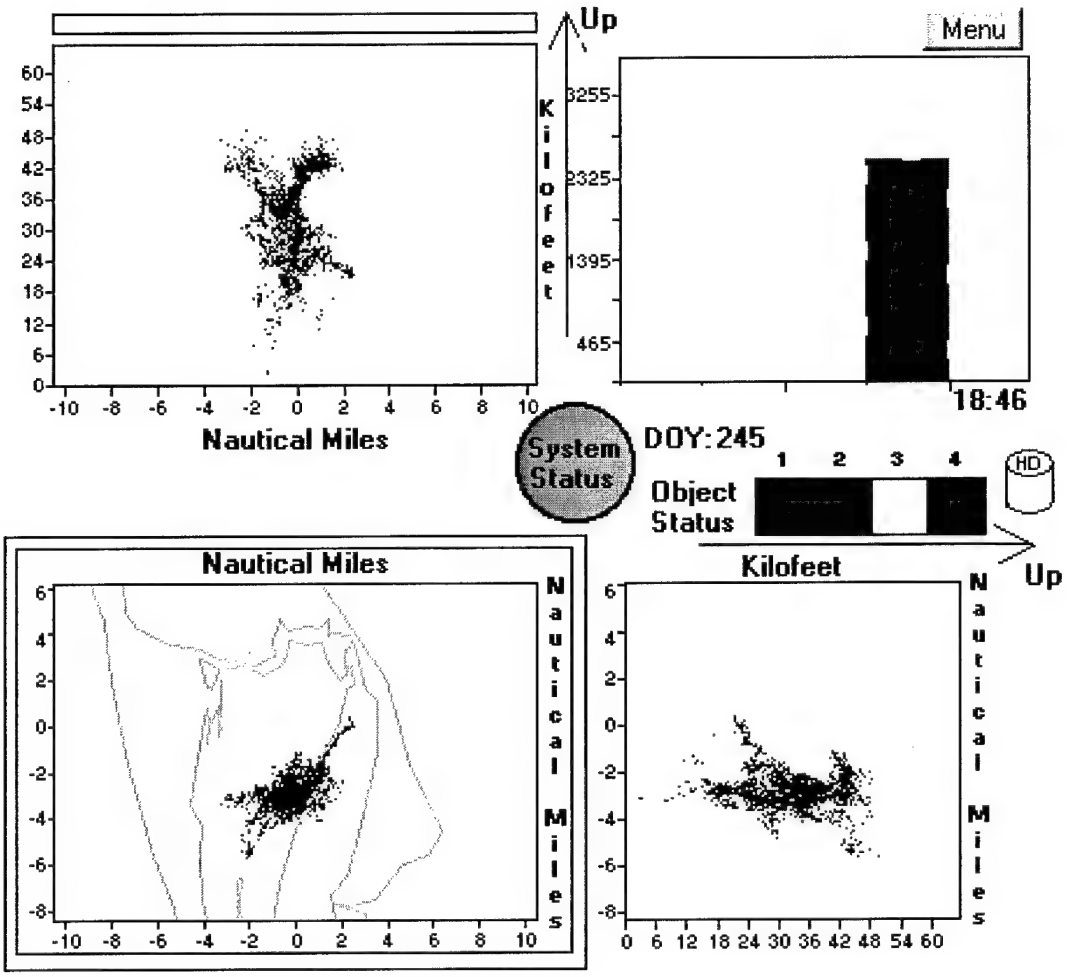

Figure 1. Example of current LDAR display. The top left quadrangle is the XZ plane, bottom left is XY plane, bottom right is YZ plane, and top right is a histogram showing activity over the past 5 minutes, with number of flashes on the vertical axis.

Data can only be visualized in 3D by mentally "folding" the three display areas into a three-dimensional box (Fig. 1). It would greatly aid the visualization process to be able to see and examine the data in a true, interactive 3D format.

1.3 Problem Statement

Although LDAR is a 3D lightning detection system, the current display is a set of 2D images. Can a true 3D visualization be developed for LDAR, and if so, will it improve the ability of the user to visualize and interpret the data?

1.4 Importance of Research

The 45 WS is tasked with providing lightning watches and warnings for the CCAS/KSC area, including space launch facilities. While the current multiple 2D display has performed adequately for the past 20 years, a properly designed 3D display will provide superior understanding of the lightning data. A 3D display can do everything a 2D display can do, since 2D is a subset of 3D, but 3D has extra advantages. The scene can be rotated and navigated through, displayed with three-dimensional perspective, and made more realistic by adding landmarks and other contextual cues into the scene.

Additionally, developing the application in a modern object-oriented language like Java will make it easier to program, maintain, modify, and expand in the future. Air Force Weather is moving toward more network-based delivery of information, including over the Internet, and a Java application can easily make the transition from stand-alone operation on a single computer to delivery over a network.

1.5 Overall Approach

This thesis consisted of two main components, processing the data and generating the graphic display. Data processing was accomplished by reading in LDAR data files one line at a time, discarding redundant data points, and writing the results to an

intermediate file. The data structures used to process the files arrange the data points into a tree. All of the individual points which would fit inside a cube 100 m on a side are "parented" by a node representing that cube. A 100 m cube, in turn, is parented by a 1 km cube which has as its children all 100 m cubes which fall within it, and 1 km cubes are likewise parented by 10 km cubes. This tree is traversed, and the coordinates of each node are written to an output file.

To generate the display, the intermediate file created during data processing is read in. The program uses it to construct a Java 3D scene graph, similar to the data structures used in data processing, which is used to render the scene. The display introduces a technique called Nested Cubes to give the user the ability to overview the entire scene, then zoom in on areas of interest and display more details. On overview, the scene will contain transparent red cubes where there is lightning activity. As the user zooms in, larger cubes will be replaced by smaller cubes that narrow down the areas where lightning is occurring, until finally the individual lightning points themselves are displayed. The Nested Cubes technique allows the LDAR data to be displayed in three dimensions, while overcoming the drawbacks inherent in 3D display.

1.6 Organizational Overview

Chapter two consists of a literature review and a discussion of the relevant theory involved in this thesis. The first topic covered is fundamental lightning stepped leader theory, followed by a general description of the operation of the LDAR lightning detection system. Next, the topic of computer visualization is discussed, with emphasis on the differences between scientific visualization vs. information visualization, and the

contrast between general-purpose visualization systems and monolithic, or single-purpose, visualization systems. The chapter concludes with discussion on computer language considerations pertinent to the thesis, including object-oriented programming, Java, and Java 3D, as well as a brief description of relevant previous work in computer visualization.

Chapter three describes the methodology used in the preparation of this thesis, including details of how the software application is designed.

Chapter four discusses the results and conclusions, and makes recommendations for future research.

2. Literature Review and Theory

2.1 Stepped Leader Theory

Although lightning discharges appear to occur as a single stroke to the human eye, in actuality a discharge occurs in a series of discrete steps, known as stepped leaders (Fig. 2). As a thunderstorm evolves, regions of positive and negative electric charge build up in the developing cloud, leading to a steady increase in the electric field. If the magnitude of the electric field increases sufficiently, naturally occurring electrons will be able to move at a velocity large enough to knock other electrons from atoms or molecules when they collide (MacGorman and Rust, 1998). This ionization causes the number of charge carriers in the atmosphere to increase dramatically. If enough ionization occurs, the air may begin to fail as an insulator, a process known as preliminary breakdown, and an electric discharge, known as a coronal discharge, can occur (MacGorman and Rust,

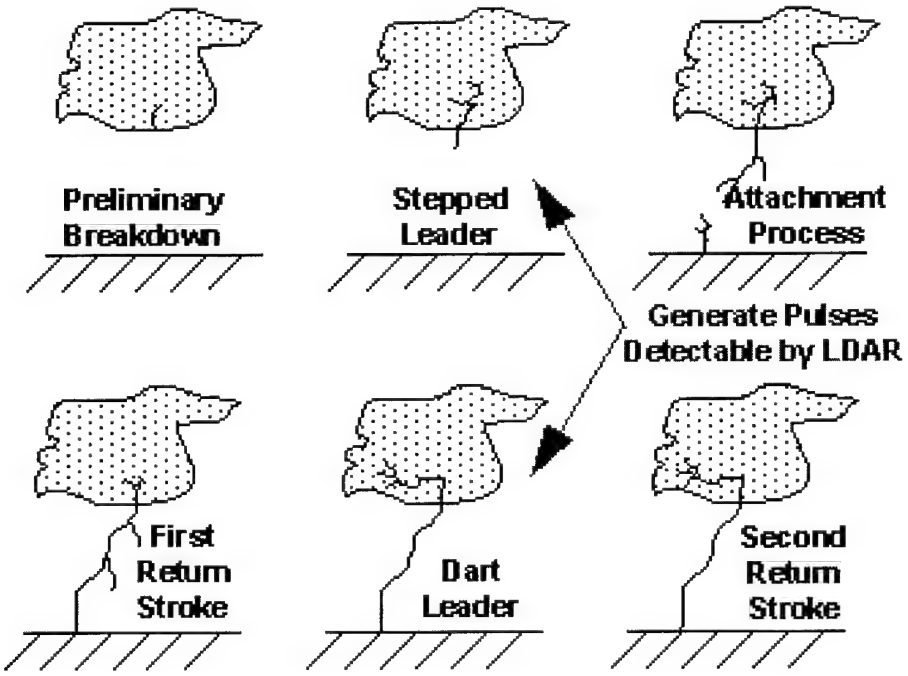

Figure 2. The lightning step leader process.

1998). This discharge, less violent than an electric spark, appears as a faint violet brushlike or branched phenomenon. These coronal discharges, which occur at the beginning of the lightning process, are known as stepped leaders. A stepped leader, typically 50 m long and traveling at a speed of 10^5 ms^{-1}, extends from a region of excess charge to an adjacent region of reduced electrical potential. Once the stepped leader has formed, there is a period of approximately 50 μs when excess charge is flowing into this newly ionized area (Krider et al., 1980). If the charge in this region, which includes both the cloud and the area newly ionized by the stepped leader, is still great enough, then the electric field strength will be strong enough for subsequent coronal discharges to occur. This process will continue until either the overall charge is reduced to the point where no further coronal discharges are possible, or the stepped leaders reach an attachment point (the ground, for example) leading to catastrophic breakdown of the insulating properties of the air, and a full fledged lightning stroke takes place (MacGorman and Rust, 1998).

In the typical cloud-to-ground lightning stroke, the lightning process starts as a series of stepped leaders starting from the cloud. When they get close enough to the ground, a return stroke is induced from the ground to the last stepped leader, completing the attachment process (Krider et al., 1980). A wave of electric charge can now flow down the ionized path created by the stepped leaders, and some of the excess charge in the cloud will be dissipated. If the cloud still contains enough charge, other strokes may occur as dart leaders retrace the path between the cloud and the ground taken by the stepped leaders. This explains the visual flickering often seen with lightning, where the frequency of individual return strokes is just at the limit of human visual perception. Dart leaders are similar to stepped leaders except they do not pause between each 50 m step,

since they follow the conducting ionized path which has already been created by the initial lightning stroke. A typical cloud to ground flash contains three or four leader-return stroke combinations (Krider et al., 1980).

2.2 The Lightning Detection and Ranging (LDAR) System

The LDAR system was developed by NASA for use at the Kennedy Space Center and Cape Canaveral Air Station, Florida. Unlike typical lightning detection systems (such as the National Lightning Detection Network), which detect only cloud-to-ground lightning return strokes in two dimensions, LDAR detects and depicts in-cloud, cloud-to-cloud, cloud-to-air, and cloud-to-ground lightning stepped leaders in three dimensions

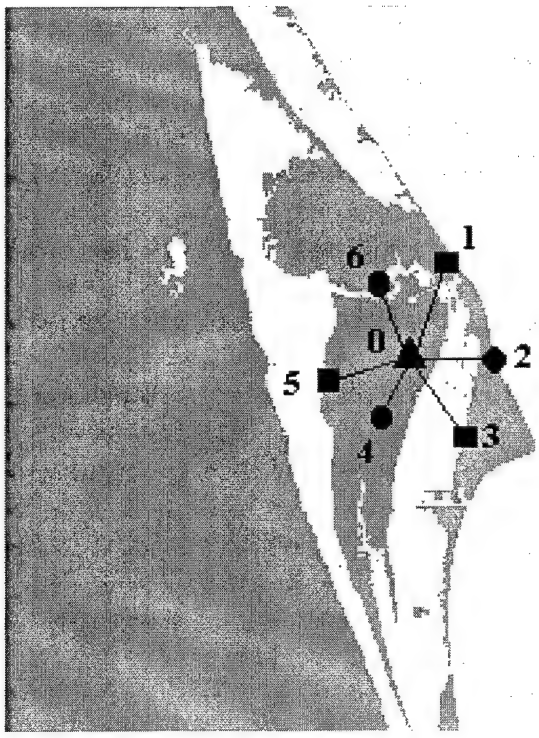

Figure 3. Map of Cape Canaveral area showing LDAR central processing site (0) and sensor sites (1-6).

(Maier et al., 1995). Developing thunderstorms usually produce in-cloud flashes several minutes before producing cloud-to-ground flashes, and thus LDAR can often warn a forecaster several minutes in advance of the threat of cloud-to-ground lightning.

When lightning stepped leaders and dart leaders occur, they emit pulses of energy over a wide range of frequencies. LDAR uses an array of 7 sensor antennas to detect electromagnetic pulses in the 66 MHz range, the middle of the frequency band which is produced during the stepped leader process (Lennon, 1979). The 66 MHz band also has the property of being relatively free of interference from other naturally occurring or man-made radio frequency (RF) sources, though false signals can occasionally be emitted from aircraft or automobile engines, or from aircraft flying through clouds or precipitation. Experienced human operators, however, can easily identify these false signals. The central processing site has one of the detection antennas, and six others are located in a roughly hexagonal pattern, all within 6-10 km of the main site (Fig. 3). When an RF pulse arrives at the central site, the processing system opens an 80 μs window for determining whether or not the pulse has the signature of a stepped leader (Maier et al., 1995), and if so, the flash is tagged with its arrival time and located in space. The location is determined by analyzing the arrival time of the same pulse at different detection sites, and using the time differences between them to plot the position of the flash. The LDAR process requires extreme time precision, and uses the Global Positioning Satellite system to provide atomic clock accuracy. In addition, the Time of Arrival (TOA) process is continually recalibrated with man-made test pulses to account for the constant minor changes of the speed of radio waves through the atmosphere from changing temperature, pressure, and moisture.

A data file is sent from the computation system to the display subsystem at one-minute intervals. This file consists of eight numerical columns per line, with each line of ASCII text in the file representing one lightning stepped leader that was detected in the interval. An excerpt from a data file is shown below:

23 16 53 32 591358 -26757 -69148 9898

where the data is in the form:

dd hh mm ss llllll xxxxxxx yyyyyyy zzzzzz

and

dd = day of the month

hh = hour of day UTC (24 hour clock)

mm = minute

ss = seconds

llllll = microseconds

xxxxxxx = E-W distance from central processing site, in meters. A positive value indicates east.

yyyyyyy = N-S distance, in meters. A positive value indicates north.

zzzzzz = vertical distance, in meters above ground level

For this project, only the last three columns, horizontal and vertical coordinates from the central site, are used. For some research applications the detailed time data may be useful, but in this case the information contained in the file name, e.g., **LDAR_Packets_990402_152005.asc,** provides all the time information needed.

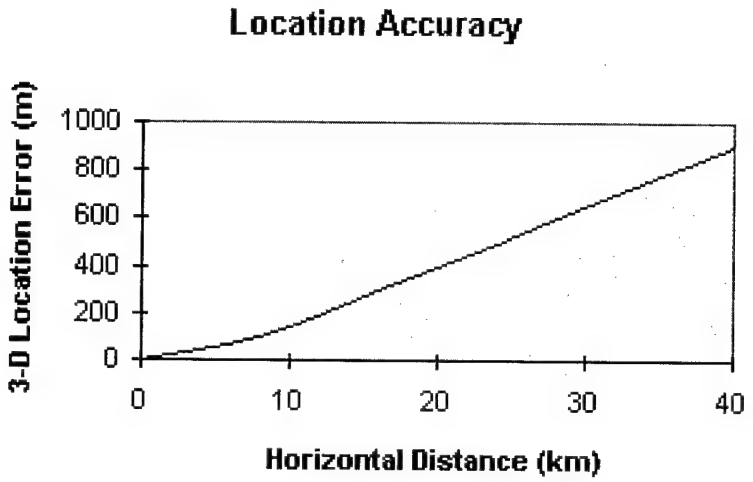

Figure 4. Accuracy of stepped leader detection as a function of distance from detector network.

The mean location error in detecting a pulse is 50 to 100 m within 10 km (Maier et al., 1995), increasing to an average of almost 1 km at 40 km from the central site (Fig. 4). Detection accuracy is greater than 99% within 25 km. It should be noted that since more than 6000 stepped leaders per second can be detected by the sensors, some events may be dropped due to overloading of the display processor (Poehler and Lennon, 1979). In addition, even when all the stepped leaders are shown, the display in the main core of lightning activity can easily become saturated, and the human viewer would not be able to discern any new data.

2.3 Visualization

2.3.1 Background. The idea of using computers to graphically display data dates back to the earliest days of the computer age, and implementations of this idea have historically advanced as fast as current hardware and software state of the art would allow

(Gallagher, 1995). Until recently, however, the considerable expense involved in owning sophisticated computer graphics equipment meant that most of the users were restricted to heavy industries and research facilities which could afford the millions of dollars required for a single computing installation. Graphical display of information on a computer is memory intensive, and until the last few years the limited size and high cost of computer memory was perhaps the single greatest obstacle to the development of visualization systems for the common consumer (Gallagher, 1995). The rapid acceptance of personal computers has brought down the price of memory dramatically, and recent advances in computer hardware and software are making the visualization of numerical data increasingly easy and affordable (Keller and Keller, 1993). Just as computer processors have followed Moore's Law (which says that processor power doubles every 18 months), graphical performance has followed a similar pattern, increasing by a power of 10 every four years (DeFanti et al., 1997). The consequence of this rapid expansion of graphical ability is that the power of visual computing can now be applied to a wider set of problems, and many scientific visualization problems can be done with commodity hardware on a PC instead of expensive graphic workstations.

2.3.2 Scientific vs. Information Visualization. Visualization can be defined as the use of computer-supported, interactive, visual representations of data to amplify cognition (Card et al., 1999). That is, the purpose of visualization is to convey insight, not produce pretty pictures. The field of visualization has generally been divided into two main areas, scientific and information. Scientific visualization, the representation of some type of physical data, is the most common and has been in use since the beginning of computer visualization. At that time, the prohibitively high cost of the necessary

graphical computer equipment limited its use to projects in industry and research which could justify the necessary large initial investment.

Information visualization, though a relatively new field, has become the subject of more and more research, using innovative tools to visualize abstract data (Andrews and Heidegger, 1998; Andrews et al., 1997; Jerding and Stasko, 1996). Unlike scientific visualization, which deals with physically based data, information visualization concentrates on the display of abstract data such as business information, data networks, databases, etc. Thus, information visualization frequently deals with data for which there is no "obvious" physical representation, and attempts to somehow use the pattern recognition abilities of the human eye to gain insight into large volumes of this abstract type of data (DeFanti et al., 1997). Information visualization has proven particularly useful for monitoring large amounts of data in real time and under the pressure to make time-based decisions, e.g., stock and commodity markets or information networks. The development of information visualization as a recognized discipline, combined with the recent advances in graphics hardware for personal computers, means that PCs are now able to support real-time, dynamic, interactive visual representations, which will open up the path for the use of information visualization in mass-market products (Card et al., 1999). In other words, information visualization is simply a new upward step in the process of using the resources of the external world, in this case via computer graphics, to increase our ability to think.

2.3.3 Monolithic vs. General Purpose Systems. Modern computer visualization systems can be grouped into two broad categories: monolithic, or single purpose systems, and more general data visualization systems. Monolithic systems, often referred to as

turnkey or end-user visualization systems, allow the user to create visualizations of his data without any programming (Gallagher, 1995) by providing a specialized interface for the user to interact with data. Many such systems are tailored to suit a particular application or related group of applications, such as fluid mechanics. The principal virtue of such systems is they are easy to use and require little time to learn, but in return for this ease of use the application generally will not be able to satisfy the requirements of all users. Rather, it is aimed at satisfying the majority of users for a particular set of tasks. General purpose systems, on the other hand, require more time and effort to learn, but give the user much more flexibility to design an application which will fit his needs (Schroeder et al., 1996). Examples of such general purpose systems are the Visualization Toolkit, or vtk (Schroeder et al., 1996), IBM Data Explorer (Lucas et al., 1992), and the Application Visualization System (Upson et al., 1989). In these types of systems, the user is provided with a set of software tools which can be linked together in order to construct an application which meets his needs (Gallagher, 1995). A common thread among general purpose systems is the ability to handle many different types of data and do a limited number of things well. The monolithic systems, by contrast, specialize in handling relatively few types of data but are able to do a greater number of things with them. Which type of system to use depends, of course, on the needs of the end user.

2.4 Computer Language Considerations

2.4.1 Object-Oriented Programming.

Within the past few years, a new approach to designing software has become very popular. Called object-oriented programming (OOP), it is fundamentally different from the procedural programming methods long used

in designing programs in Fortran, Pascal, or C. In object-oriented programming, software designers build modules of code, called objects, that accomplish very specific functions. Programmers are able to combine these objects to build custom applications, and can easily recombine the objects to update applications, add new features, and make them easier to use (Schroeder et al., 1996). The benefit of OOP is that it increases the rate of software development, allowing more powerful and customized applications to be built in less time (Eckel, 1998).

The two dominant object-oriented programming languages in use today are C++ and Java. C++ is more mature as a programming language, having been in use much longer, as well as having a large number of preexisting C programmers to draw upon. Java, by contrast, has only been in existence since 1995, though its popularity has increased markedly since its introduction. Making the transition from C to Java is not that difficult, since Java borrows much of its syntax from C (Eckel, 1998). The main differences between C++ and Java have to do with the fact that C++ had to be constructed to be backward compatible with C, which means some compromises in "object-orientedness" had to be made. Java's designers, on the other hand, started with a blank slate and designed Java from scratch to be an object-oriented language (Horstmann and Cornell, 1999).

2.4.2 The Java programming language. The Java programming language, developed by Sun Microsystems, was released in 1995 and quickly gained popularity as the programming language of choice for the Internet (Elvins and Hibbard, 1998). While Java does enjoy many advantages, it has some drawbacks. The main disadvantage of Java as a programming language has been its slow execution speed. In most cases, a

computer application is compiled once into executable code for running on a computer. Java is an interpreted language, which means that instead of being compiled into machine language, a program is translated into an intermediate form called bytecodes. The machine on which the program is running contains an interpreter which translates the bytecodes into machine language, which can then be executed on the computer (Schroeder et al., 1996). This translation occurs step by step during every run of the computer program, as opposed to a compiled program, which is translated just once. Inserting this intermediate step causes a Java program to run slower than a compiled program, by as much as a factor of ten (Horton, 1997), a definite disadvantage in comparison to compiled languages like C++. Most of this speed disadvantage can be mitigated, however, through the use of a just-in-time (JIT) compiler. JIT compilers convert Java bytecodes into machine instructions as they are loaded. Programs will take a little longer to load, but once loaded they will run at virtually the same speed as a regular compiled program (Eckel, 1998).

Interpreted languages do have an important advantage, though, and this advantage is the primary reason for the explosion in popularity of Java. While it is true that a compiled program runs faster than an interpreted one, a compiled program can only run on one specific type of computer platform. That is, a given program must be compiled separately on each type of machine before it will run, which essentially means that some type of modification to the program is generally necessary for each platform. An interpreted program, on the other hand, has the advantage of needing to be written only once. The interpreter on each machine handles the machine-specific instructions needed to make the program run on each separate platform. This is the reason why Java is often

referred to as a platform-independent programming language. A Java program needs to be written only once, and it will work on any computer platform which has a Java interpreter (Horstmann and Cornell, 1999). This platform-independence has made Java the programming language of choice for implementing network services, particularly on the Internet, since an application written in Java can be downloaded and run on any computer (Horstmann and Cornell, 1999). Equally important, though less often stressed in all of the hype surrounding Java, is reduced complexity for the programmer. Since Java was designed starting with a clean sheet of paper, it was able to avoid many of the programming complications inherent in other languages. This has resulted in reductions in development time by as much as a half compared to the time it takes to create an equivalent C++ program (Eckel, 1998).

2.4.3 Java 3D. Java 3D is an application programming interface (API) used for writing three-dimensional graphics programs. It is a high-level graphics programming interface, designed to take advantage of Java's "write once, run anywhere" portability across computer platforms while allowing for easy and efficient development of large virtual worlds (Sowizral and Deering, 1999). Java 3D uses calls to either OpenGL or DirectX, the graphics standards developed by Silicon Graphics Incorporated and Microsoft, respectively, to do the actual low-level rendering to the screen (Brown and Petersen, 1999). Of the two, OpenGL is the most popular industry standard and is supported by almost all of the common graphics hardware being produced today. The OpenGL version of Java3D was used for this thesis. In the following discussion of Java 3D, names of objects, such as BranchGroup and Shape3D, are capitalized in accordance with the standard Java convention.

The Java 3D API is object-oriented, just like the Java programming language itself. A graphics application is built in Java 3D by constructing individual graphics elements as separate objects, then connecting them together into a treelike structure called a scene graph (Fig. 5). This scene graph, commonly called a virtual universe,

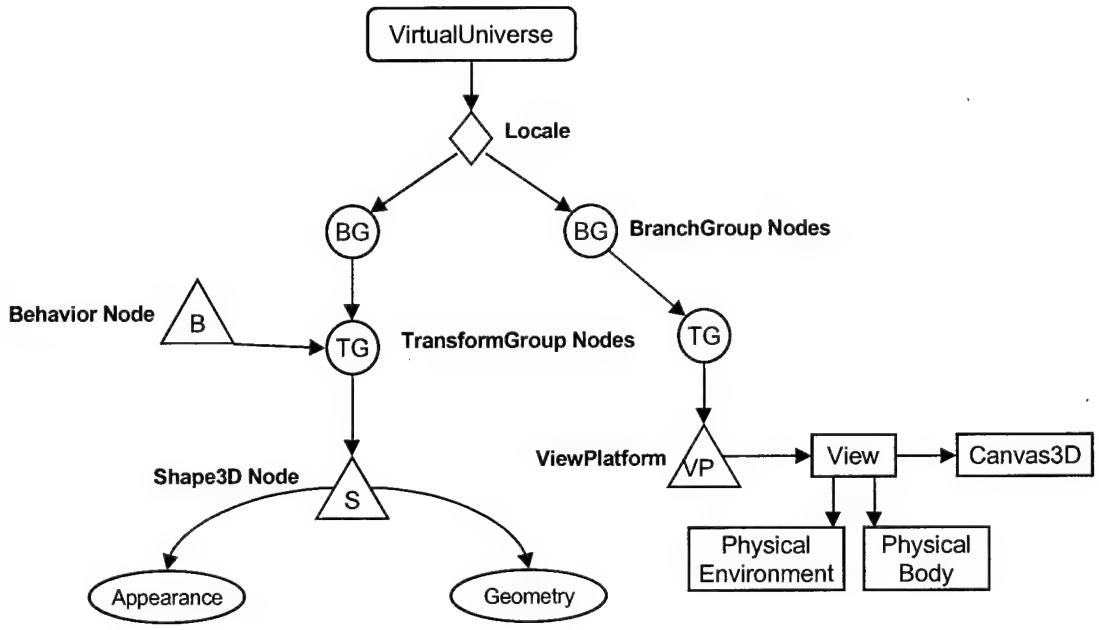

Figure 5. Example of a Java 3D scene graph.

contains a complete description of the entire scene. Unlike many other graphics APIs, such as OpenGL, all of the low-level details involved in rendering the scene are taken care of by the scene graph programming model, and the developer can just concentrate on composing the scene (Sowizral et al., 1998).

The scene graph shown in Figure 5 illustrates the basic elements which make up a Java 3D virtual universe. At the root of the graph is a VirtualUniverse object which contains and encapsulates the entire scene graph. The VirtualUniverse object contains at least one Locale object, which is essentially the origin of the virtual universe. Normally there is only one Locale object, but there are reasons why a programmer might choose to have more than one (Sowizral et al., 1998). For instance, if a project involved modeling two detailed scenes which were widely separated geographically, e.g., Andrews AFB, MD, and Vandenberg AFB, CA, in the same scene graph, it would be much easier to program if two separate Locales were used, one centered at Andrews, one at Vandenberg. For most applications, however, one Locale will do.

Attached to the Locale object are one or more BranchGroup nodes. Note that in general the term node simply refers to any Java 3D object in the scene graph tree. Group nodes exist as a way to organize the structure of the scene graph. A group node can have zero or more children, but only one parent. BranchGroups do not actually perform any operations themselves, but rather serve as general-purpose grouping nodes (Brown and Peterson, 1999). The arrangement shown in Figure 5 is typical of most scene graphs in having two basic BranchGroups attached to the Locale. The left branch represents the objects contained in the virtual universe, and the right branch represents the viewer, which controls the point of view from which the scene will be observed. The viewer can be likened to a camera mounted on a platform in the virtual universe, a platform which can be moved around, tilted, and aimed independently from the actual objects in the universe.

TransformGroup nodes are like BranchGroups in that they can group other nodes, but they do more than just serve as attachment points for other nodes. They perform transformations (translation, scaling, and rotation) on all of the children below them in the scene (Sowizaral et al., 1998). This principle is key to understanding how the scene graph model operates. A given node operates on its children, and its children's children, and so on down the line. As an example, the TransformGroup on the left side of the scene graph in Figure 5 controls the location of the Shape3D node below it, and the TransformGroup on the right side controls the ViewPlatform below it, as well as everything attached below the ViewPlatform.

Finishing the discussion of the left branch of the scene graph in Figure 5, there are two objects directly attached to the left TransformGroup. The object directly below is a Shape3D node, which as its name suggests, represents a physical object in the virtual universe. In fact, a Shape3D node is the only type of object which directly appears in a scene, and the rest of the scene graph structure can ultimately be thought of as simply supporting structure for properly displaying the Shape3D node (Sowizral et al., 1998). The two ovals attached to the Shape3D node, Appearance and Geometry objects, are two examples of such supporting objects. Geometry contains the list of points which make up the object, as well as supplemental information about its basic geometry, while the Appearance node controls the color, texture, and other aspects of an object's appearance. The reason that attributes like appearance and geometry are kept as separate objects instead of being incorporated into the Shape3D node is to make each object as reusable as possible (Brown and Peterson, 1999). For example, a single Appearance object can be used to define the outward appearance of any number of objects, irrespective of their

geometry, and a given geometry object need be created only once, then referred to each time a copy of that particular 3D shape is needed. This is also a good example of one of the principles of object-oriented design—reusable modules of code. The last remaining object on the left branch of the scene graph is a Behavior node, attached to the TransformGroup. The function of Behavior nodes is to provide a means for modifying the scene graph while the application is running. Rotation of objects, turning parts of the scene on or off, or moving objects around in the scene illustrate some of the uses of Behavior nodes.

Having covered the left branch, which depicts the objects to be modeled within the virtual universe, we now look at the right branch in Figure 5, which describes how that universe will be viewed. Remember the right TransformGroup controls the location and orientation of the ViewPlatform object, which can be thought of as a platform with a camera mounted on it. What this camera sees is represented by the attached View object, which controls parameters like field of view and depth of the scene, and then this view is displayed on a Canvas3D object, i.e., a window on the screen. There are two other objects attached to the View, PhysicalEnvironment and PhysicalBody, that are used to furnish further information about the viewer. Such information might include the distance between the viewer's eyes for stereoscopic vision, or the distance of the viewer's head above the ViewPlatform (Brown and Peterson, 1999). These considerations may come into play when designing a complex, interactive virtual world, but in most applications the default versions of these objects are used.

The previous discussion is intended as a brief orientation to Java 3D and the scene graph model. The subject is, of course, considerably more complex, and the reader is referred to Sowizral et al. (1998) for a more complete description.

2.5 Previous/Related Work

The data flow paradigm has become a popular architecture for scientific visualization (Pang and Alper, 1994; Wood et al., 1996). The data flow model treats the visualization process as a "pipeline" in which data is fed in, operated on by modules in the pipeline, and flows out the end as a final image. The term "data flow" itself refers to the production and consumption of data as it flows through a network (Pang and Alper, 1994). As data moves from one module to another, each module in the network operates on the data in some way, filtering (changing the form of the data), mapping (from raw data to some form which is capable of being visualized), or rendering (drawing the picture). IBM Data Explorer (Lucas et al., 1992), the Application Visualization System (Upson et al., 1989), and vtk (Schroeder et al., 1996) are examples of popular visualization environments which are designed around the data flow model. In the realm of meteorological visualization, environments such as VizWiz (Michaels and Bailey, 1997), Vis5D, and VisAD (Elvins and Hibbard, 1998), a Java-based version of Vis5D, also use the data flow paradigm. Of these, Vis5D has long been a favorite for three-dimensional visualization in the weather research community. It is a general purpose visualization system, as opposed to a monolithic or single purpose visualization system, and though useful for atmospheric model assessment and analysis, it has often been considered too general purpose for use in the operational forecasting environment

(Treinish, 1998). Such highly generalized systems are often capable of providing functionality similar to that of the single-purpose systems, but the learning time involved in mastering a general purpose system is often considered unacceptable for time-critical activities like weather forecasting.

It has long been recognized that there are some specific challenges in the modeling of environmental data. Such data frequently consists of unevenly distributed samples (Xiao et al., 1996), and since virtually all data flow models require gridded data in order to do isopleths and contouring, data visualization applications must interpolate their samples onto uniform 2D or 3D grids before they can be rendered. Another important factor in dealing with environmental data is the size of the data sets involved (Star, 1993). The designers of VizWiz, a Java-based visualization application, found that even when working with advanced graphic workstations, memory and processor limitations set a practical upper limit of fewer than 125,000 points for data sets (Michaels and Bailey, 1997). Personal experience has also shown, while constructing a C++ application to display LDAR lightning data for a previous visualization project, that it is difficult to work with data sets larger than approximately 150,000 points.

3. Methodology

3.1 Overview

The purpose of this research was to develop a 3D display system for LDAR data, using the Java programming language to take advantage of modern graphics hardware and software. The display would be runnable from any type of computing platform that supports Java, and could easily be modified to deliver lightning data over a network, including the Internet. First an application, called LConvert, was developed to take the raw LDAR data and process it into an intermediate file format for later ingestion by the display application. Then another application, LViewer, was developed to read data from this intermediate file and construct a 3D scene which can be fully navigated by the user. The design of the display incorporates a technique of "Nested Cubes" to implement the information visualization methodology of Overview, Zoom, then Details on demand.

3.2 Taxonomy

The underlying design of a visualization application should be based on its purpose. There have been several attempts in the field of information visualization to develop a general taxonomy to determine the correct type of visualization to use for a given situation, based on the type of data involved and the types of tasks which will be performed on the data. Schneiderman (1996) proposed a taxonomy based on grouping data into seven basic types (1-dimensional, 2-dimensional, 3-dimensional, Temporal, Multi-dimensional, Tree, and Network) and seven types of tasks (Overview, Zoom, Filter, Details-on-demand, Relate, History, and Extract). Keller and Keller (1993), on the

other hand, proposed a more abstract process of first identifying the visualization goal, removing mental roadblocks, then deciding between data or phenomena as the focus of the visualization. Given that the data used in this application consists of a series of coordinates representing 3D points in space, it would clearly fall into the three-dimensional data type of Schneiderman's taxonomy. In order to further decide how to visualize this particular type of data, one must first make some assumptions about what the goals for the visualization will be. In this thesis, the goal will be defined at the outset as designing a three-dimensional display system for LDAR data.

The most basic way to display the LDAR lightning data would be to simply read in a raw LDAR file, pick out the x, y, and z coordinates for each recorded stepped leader, and then display the data as one big group of points. This approach has the advantage of simplicity and would lend itself easily to displaying the data in a time series. If one wanted to examine lightning data over the past five minutes, for example, then each data file could be read in sequentially, and each group of points displayed with a different color to represent each separate time step. This is how lightning data is currently treated in traditional 2D displays, and while simply expanding this type of display from 2D into 3D would yield some improvement, it can also cause new problems. The viewer in a 3D display must have an understanding of his position and orientation in order to correctly interpret the objects in the display, and there is also the serious problem of occlusion (Schneiderman, 1996). In a 3D lightning display which represents stepped leaders as disembodied points floating in the air, it is possible to "lose" some points which may lie behind another dense group of points (occlusion), or the user may not be able to distinguish between points which are in the foreground from those in the background

scene at some angles. Therefore, one cannot simply display individual points in three dimensions without taking these potential problems into account.

Given these limitations, the question becomes one of how to display the data effectively, using the advantages of 3D while minimizing the drawbacks. To answer this question, one can look to the developing field of information visualization. At first glance, the problem of displaying three-dimensional lightning data seems like a straightforward case of scientific visualization, since the data is physically based and there is a direct spatial mapping from data to the visualization environment (DeFanti et al., 1997). Strictly speaking, however, scientific visualization can be described as taking raw numbers and turning them into something visually meaningful (Card et al., 1999). Common examples of this include contour mapping, isosurfaces, and isopleths, which transform a grid containing nothing but raw numbers into a useful visual representation from which the user can gain insight. In the case of lightning visualization, however, the focus is not on the numbers (coordinates) contained in the data, but instead on where the data points are located. This would suggest that what is truly being visualized in this case is not data, but rather the phenomenon of lightning itself. Keller and Keller (1993), in their methodology for selecting visualization techniques, make this distinction by saying that data representation shows the data values independent of the phenomenon, and the viewer must deduce the relationship of the data to the phenomenon. If the focus is on the phenomenon itself, however, then contextual-cue techniques (landmarks, backgrounds, etc.) are used to provide clarification and interpretation to the representation of the data.

For this thesis, it seems appropriate to consider the phenomenon itself (location of lightning stepped leaders) as the main emphasis, because LDAR was designed as a

lightning warning system. Its purpose is to define the existence and the volumetric extent of the lightning hazard, not to get an exact count of the number of stepped leaders (Maier et al., 1995). In other words, LDAR is designed primarily to accurately tell the user where lightning is occurring. Therefore, the goal may now be further refined as: Designing a three-dimensional display for LDAR data which shows the location/volume where lightning is occurring.

3.3 LViewer Design

Having defined the goal in terms of the phenomenon, the mechanics of actually implementing it can now be considered. Referring back to Schneiderman's (1996) taxonomy for designing user interfaces, he presents what he terms the Visual Information-Seeking Mantra: Overview first, zoom and filter, then details-on-demand. In applying these principles to the display of LDAR data, one can now begin to describe how an ideal visualization application would perform. First of all, consider the case in which there is one single stepped leader in the entire visualization. The total volume being visualized is on the order of tens of thousands of cubic kilometers, but the user needs to be able to clearly identify where the single stepped leader is when looking at an overview of the entire area. Next, take the case where there are one or more large groups of stepped leaders, but with a single point lying outside of the large group(s), possibly even behind them, which the user still needs to be aware of. These situations would suggest working toward a solution where a single data point can readily get the user's attention, even during an overview of the entire scene, or when numerous other points may threaten to obscure the presence of isolated outliers.

The "Nested Cubes" approach was developed to deal with the problems involved in visualizing a large lightning warning volume. The area around Cape Canaveral is divided into a series of non-overlapping cubes 10 km on a side. If any stepped leaders are detected within a given volume, they become visible in the scene as a transparent, red 10 km cube. This gives the users an overview of the entire Cape, yet at the same time alerts them to the presence of lightning in any specific area. If the user needs more detailed information about lightning in a particular region, she can then zoom in toward a given 10 km cube. When the viewer is close enough, the 10 km cube will disappear and be replaced by one or more transparent, red cubes 1 km on a side, which bound the areas where lightning was detected inside the original 10 km cube. As the viewer approaches these smaller cubes, they will in turn dissolve into still smaller cubes, 100 m on a side, bounding the areas with lightning. Finally, if the viewer continues to zoom in, the 100 m cubes will be replaced by individual points of 1 m resolution, which are the actual points found in the original LDAR data file. One should also keep in mind that beyond approximately 8 km from the central processing site, the mean error of the LDAR system in locating a stepped leader is greater than 100 m (Fig. 4), so that as one approaches this distance, even the 100 m cubes may not be a totally accurate representation of where lightning activity is occurring.

Each of these transparent cubes is independent of the others. For example, if a given 10 km cube breaks up into three 1 km cubes, each of these cubes will dissolve into 100 meter cubes one at a time, as the viewer approaches. It is also worthwhile to note that there is nothing magical about the selection of 10 km, 1 km, and 100 meters for the sizes of the cubes. Experimentation with different cubes showed that making the sizes

differ by a factor of 10 produced a good visual effect. By the time the viewer has zoomed in enough on a particular 10 km cube so that it begins to fill the screen, it will start to break up into smaller cubes. Having a 10 meter cube intermediate between the 100 meter cubes and individual points was tried and discarded, as it seemed to make the scene too busy and did not appear to make the visualization any clearer. In addition, earlier versions of the application experimented with using spheres instead of cubes to bound the lightning areas. However, if one considers a given point, which lies near the edge or corner of a cube, then what size sphere would be best to contain that point? A sphere which would lie totally inside the cube would not include points near the edges or corners, while a sphere big enough to surround the entire cube would have the disadvantage of overlapping adjacent spheres. Using cubes proved to be the better choice, since cubes can be stacked and placed adjacent to each other in such a way as to cover the entire volume without overlap. It is also much easier to mathematically determine if a given point lies inside a cube, as opposed to a sphere.

The "Nested Cubes" approach allows the user to perform the basic information visualization functions of overview, zoom and filter, and details on demand while mitigating the possible drawbacks of 3D viewing as detailed in section 3.2. The problem of occlusion is avoided by making the cubes transparent. A large, dense group of points in one area will not be able to block out other lightning activity behind it in the scene, since other, more distant activity will still be clearly visible as a red cube in the background. Likewise, the possibility of losing a single data point in the scene is reduced through the use of cubes to display general areas of lightning activity.

3.4 Data Processing

The raw LDAR data for this project was obtained from the Global Hydrology Resource Center web site. Its format is described in section 2.2. Originally, the LViewer application was designed to read LDAR data directly from these raw data files and create the display. This approach worked for the majority of data files, which are on the order of a few hundred thousand bytes or less. But when dealing with large data files, greater than about two megabytes (Mb), the application would run out of memory. This problem was thought to be caused by the need to hold a given set of data in memory twice, in effect. First, the entire file had to be read in, processed to eliminate duplicate points, and the data stored in some type of data structure. Then, the program would traverse the data structure and build the appropriate scene graph for display on the screen.

Clearly, some way was needed to reduce the amount of memory required to work with a given file. Reducing the amount of memory needed by the Java 3D scene graph would be hard to do without eliminating essential features of the program. It would also be difficult to reduce the size of the data structures needed to process the file, since execution of the program depended on having a data structure which closely mimicked the structure of the scene graph. The other option was to use some type of intermediate file to store the data from the data structures, which could be read in as needed to construct the scene graph, eliminating the need to hold everything in memory twice.

Using an intermediate file was the option which seemed most likely to succeed, and the Java programming language contains a method called object serialization that greatly simplifies the process of writing to a file. Object serialization saves an object's state in a series of bytes so that the object can be reconstituted from those bytes at a later

time (Harold, 1999). The serialized objects can then be written to any output stream (a file, for example). In other words, the programmer can save an object and read it back in without worrying about all the details which would normally be involved in making sure an object's state is stored correctly.

This object serialization procedure was implemented to store the data structure objects in a file, with the intention of reading it back in as needed to construct the scene graph. However, it proved to be of little help in solving the insufficient memory problem. In order to construct the scene graph correctly, one would have to read virtually the entire data structure back into memory, and in the end wind up in exactly the same position of having to hold everything in memory twice.

The final solution involved the use of a predefined data format which could be saved into a file, then read back into the program one object at a time. The original LViewer application was thus split into two parts: LConvert reads in the raw LDAR data file, processes it, and writes the results to an intermediate file; LViewer reads in the data and constructs a scene graph with it. Both steps could have been accomplished within a single application, but breaking the process up into two parts had a distinct advantage. Using LConvert to do the initial processing would greatly reduce the size of the files needed to hold the LDAR data (explained in more detail in section 3.5), requiring much less bandwidth to send the data over a network in the future. The next two sections present the details of the LConvert and LViewer applications, respectively.

3.5 The LConvert Application

LConvert is the application which reads in a raw LDAR data file, processes it to remove redundant data points, and writes the results to an intermediate file which can be read by LViewer and used to construct a scene graph. To explain how it works, it is instructive to follow the path taken by a single stepped leader as it makes its way through the program.

Each line in an LDAR data file represents one stepped leader detected, and consists of eight numbers. The x, y, and z coordinates are the sixth through eighth numbers in each line (section 2.2), so LConvert skips the first five numbers and reads the last three into a temporary array. The first step in processing a given data point is to determine if all of the "Cubes" needed to contain that point already exist (section 3.3), and if some or all of them do not exist, create them.

When the first point is read in from the file, the x, y, and z coordinates for the point are used to determine if a Cube10km object which can hold this point already exists. The data structures in LConvert are all versions of a Hashtable, a type of data structure which allows for rapid searching based on a hashcode. A hashcode is a way to take some information in the object in question and turn it into a relatively unique integer for that object (Eckel, 1998), which is used to perform a quick search. Then, instead of doing a linear or binary search for the object, the hashcode can be used to rapidly locate the object. The hashcode for this point is calculated by isolating the ten thousands digit of the x, y, and z coordinates (e.g., the 3 in 31456), multiplying each of them by a prime number, and adding the results together (detailed in Appendix K). This generates a unique hashcode with which the Cube10km that should contain this point can be

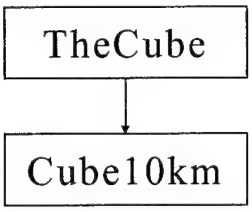

Figure 6. TheCube object containing a single Cube10km.

identified. The result is that, when other points are read in, each point which would be physically located inside of that particular Cube10km will also have the same hashcode, allowing those points to also be placed inside the same cube. So, in this case, the hashcode based on the point is calculated, and the Cube10km which would contain the point is searched for. Since this is the first point in the file, the search will come up negative, and the Cube10km will have to be created. This Cube10km is placed into its parent data structure, TheCube, which contains all of the Cube10km objects. At this point, the data structures in LConvert look like Figure 6, with only one Cube10km inside TheCube.

Next, the process is repeated to see if the correct Cube1km object exists for the first point. This time, the thousands digit of the coordinates is used to calculate the hashcode, and the proper Cube1km is searched for. Again, the search is negative, and a Cube1km object is created and placed into the Cube10km hashtable, as shown in Figure seven.

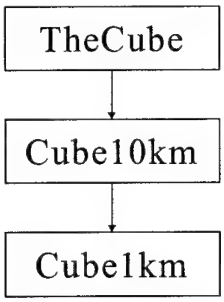

Figure 7. A Cube1km object is added to the data structures from Figure 6.

Again, the process is repeated for the Cube100m object, using the hundreds digit to calculate the hashcode, and a new Cube100m object is created and placed into the Cube1km hashtable, as shown in Figure 8. Then, the Cube100m hashtable will be searched to see if it contains this particular point. Of course it does not, and so that point will be added to the Cube100m hashtable. Figure 8 shows what the data structures in

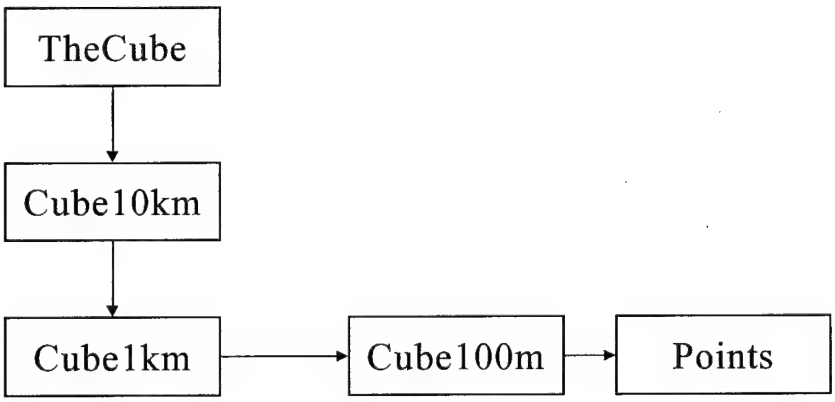

Figure 8. LConvert data structures after one point has been processed.

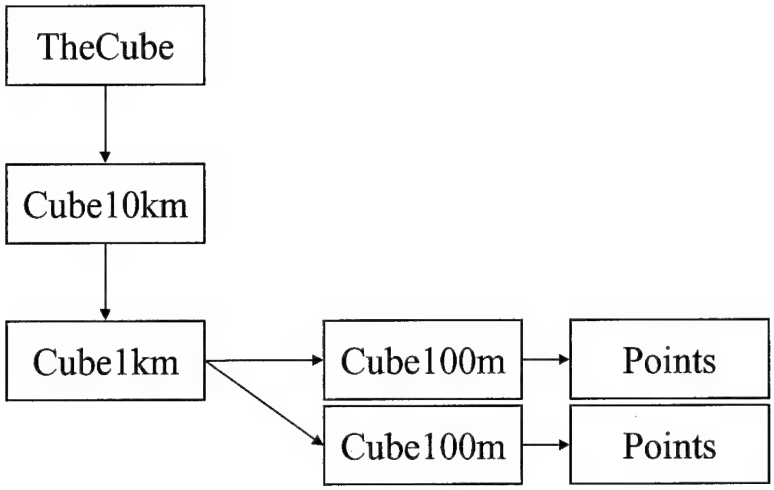

Figure 9. LConvert data structures after a second point has been added.

LConvert look like after one point has been processed.

Now suppose the second point in the data file is read in, and this point is a few meters away from the first point. It still falls within the same Cube10km and Cube1km objects, but is not close enough to fall within the same Cube100m object. The resulting data structure is shown in Figure 9. Note that the "Cubes" used to contain the data points start from the origin, the LDAR central processing site, and cubes of the same size do not overlap. Viewed from above, adjacent cubes of the same size would form a checkerboard pattern with a cross over the origin. That is, the Northwest 10 km cube would extend from (0,0) to (10000, -10000), the Northeast cube would extend from (0,0) to (10000, 10000), and so on.

Successive data points are read in and processed until the entire file is done. Duplicate points are discarded, since there is no benefit in displaying the same point twice. It would be simple to modify the program to keep track of the number of times

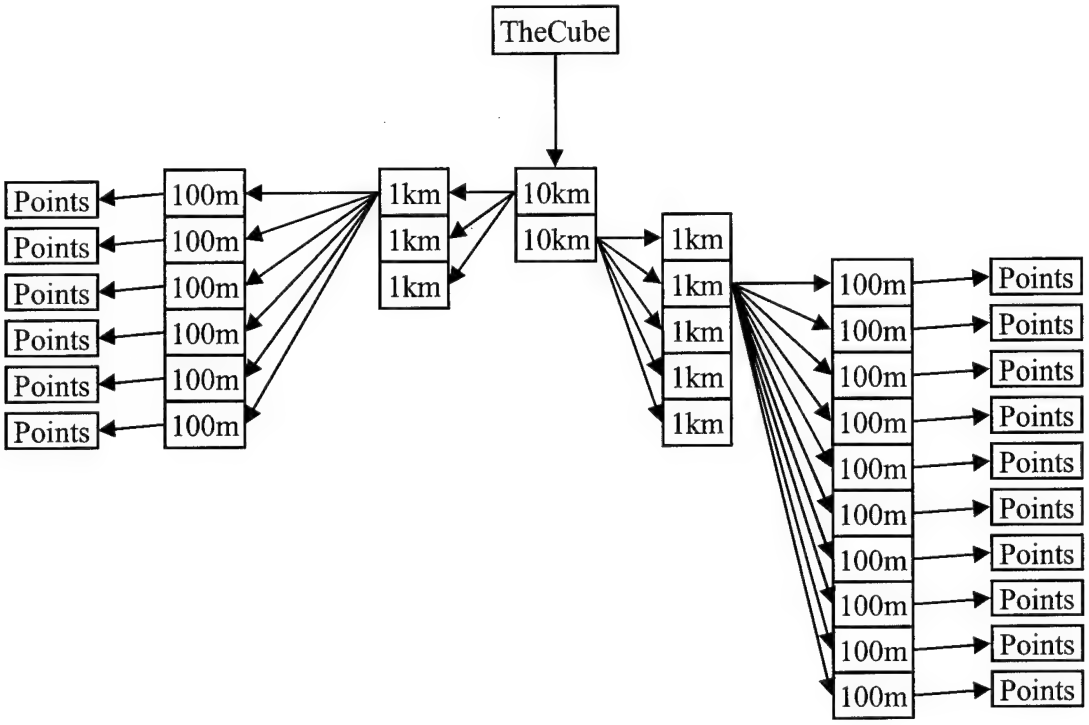

Figure 10. Example of a full LConvert data structure. Names are shortened, so that Cube10km is shown as 10km, Cube1km as 1km, etc.

each point is encountered, by modifying the MyPoints object to include a counter, if one wished to do so. Figure 10 shows an example of how the LConvert data structures might look after processing a large file.

Once the data has been processed and stored in the data structures, the now-condensed information can be written to a file. All numbers are written as regular 4-byte integers, in binary format, which allows numbers in the range of ± 2,147,483,648 to be represented. This is more than enough, considering that the maximum magnitude for the coordinates in the data is less than ± 100 thousand, and so this extra range in the 4-byte integer can be used to encode additional information. The writeContents method of the LDataConverter object does the actual writing to the output file. It traverses the treelike

data structure in the LConvert object (Fig. 10), saving the x, y, and z coordinates of each object as ordinary integers. The first object it will encounter is a Cube10km, and the coordinates of this first cube's center point are written to the file. This is where the encoding of extra information comes in. When this intermediate file is read in by the LViewer program, it will be necessary to identify each set of coordinates as belonging to one of the cube types or as a point. Since the range of coordinates will seldom, if ever, exceed 100 thousand, 3 million is added to the x coordinate value for a Cube10km, 2 million for a Cube1km, 1 million for a Cube100m, and nothing is added to a point. When the data is read back in, a simple check of the range of the x coordinate will let the program know which object it is dealing with, and the scene graph can be constructed as the data is read in one line at a time.

The raw LDAR data files were in ASCII format, so between cutting out the unneeded data in the original files and saving the results in binary format, the size of a given file can be reduced. For example, files on the order of a few hundred thousand bytes are usually reduced by a factor of ten, or 90%, while files on the order of megabytes are generally reduced by half. This disparity seems to be due to the fact that in the smaller files, the lightning activity is occurring at only one or two locations, and processing the files removes many duplicate points. In the larger size files, the lightning activity is very widespread, so that even after removing duplicates there are still large numbers of cubes and data points to be displayed, and the reduction in file size is not so dramatic.

Even after a reduction in size by about a half, it still proved impossible to process files larger than about 4 megabytes. While this inability to process the larger files is a

drawback, it was still an improvement over earlier efforts, using the C++ programming language, which could not process a 2 megabyte file. This problem is likely due, in part, to some of the inherent drawbacks associated with object-oriented programming. In one sense, the development of new applications is made easier, since the programmer is drawing on an existing library of well-designed objects (Eckel, 1998). On the other hand, using objects which are designed by someone else means the programmer has less control over the detailed inner workings of the program. LConvert, for example, makes use of some relatively complex data structures, like HashMaps and HashSets, to keep track of cubes and points. While this makes the program easier to write, the programmer has little or no control over how much overhead these objects will generate, and, as a result, the available system memory may be exhausted. The alternative is to build one's own objects, and have more control. This would undoubtedly allow larger LDAR data files to be processed, since the hashtable objects in Java are over-engineered for what the LConvert application needs.

3.6 The LViewer Application

3.6.1 The LViewer Scene Graph

3.6.1.1 Main Scene Graph. This section describes the design of the overall main scene graph in the LViewer application. In the Java 3D API, the scene graph contains a complete description of the entire scene, or virtual universe, that is being modeled (Sowizral et al., 1998). The details of actually rendering the virtual universe to the screen are taken care of by the API, allowing the developer to concentrate on scene

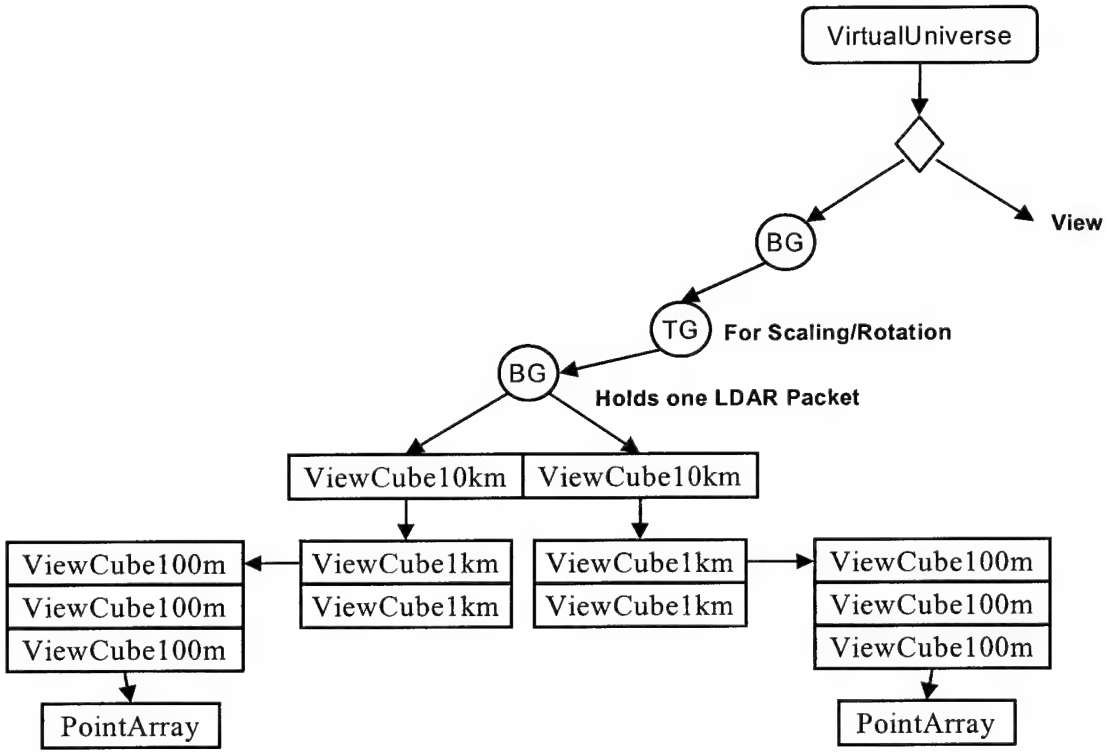

Figure 11. The LViewer scene graph.

graph design.

The LViewer scene graph is shown in Figure 11. Note that from the lower BranchGroup (BG) on down, the graph closely resembles the LConvert data structures as shown in Figure 10. The LViewer and LConvert applications were designed with this feature in mind, so the LViewer scene graph could be constructed step for step as one traversed the LConvert data structures. The various ViewCube objects near the bottom of the graph are explained more fully in the next section.

The basics of general scene graph construction are covered in section 2.4.3. In the Virtual Universe shown in Figure 11, the Locale (diamond shape near the top), or the "origin" of the Universe, is centered on the LDAR central processing station located at 28.5387° N, 80.6428° W. Since the LDAR data is given in terms of meters from the central processing site, this makes for easy placement of the detected lightning stepped leaders.

The next object of interest in Figure 11 is the TransformGroup (TG) about halfway down the scene graph. The function of a TG object is to change the position, orientation, and scale of the objects below it (Sowizral et al., 1998). In this case, all of the methods for navigating through the scene using the mouse operate on this TG (more in section 3.6.2). The TG is also used to control the size of the scene through scaling of the objects below it. All of the calculations for placement of cubes and stepped leader points are done using the actual measurements in meters, as are those for the ground surfaces and any physical objects in the scene. So the Virtual Universe is actually modeled at full scale, covering an area of 80 km by 80 km. Then the TG is used to scale the entire scene down to a manageable size before it is rendered to the screen. The

absolute size of the scene doesn't actually matter for display purposes, but it does make a difference when navigating through the scene. If the Virtual Universe is scaled down to an extremely small size, the viewer will move through the scene too rapidly, and if it is scaled too large (such as leaving it at full size in this case) navigation within the scene will be too slow.

There is only one BranchGroup, and therefore one branch, attached to the TransformGroup. This represents the data contained in one LDAR file. If data from more than one time period were to be displayed, it would probably by done by adding a separate branch to the TG, though this is not the only way to accomplish it. In its present form, LViewer can only display data from one time interval. Adding data from additional time intervals is possible, but would require more complex data processing to integrate with the "Nested Cubes" paradigm. In one approach, each point from a second file would be compared with those from the first, to see if it is already contained within a preexisting cube. If so, then it would be added to that cube, presumably using a different color to differentiate it from the original points. If not, a new cube would be created, again using a different color to distinguish it from cubes belonging to the first set. For a third set, the process would have to be repeated for the two preceding sets, and so on.

3.6.1.2 The ViewCube Object. The ViewCube objects near the bottom of Figure 11 are the key to controlling the display of the various cubes and points within the scene, so their inner workings are covered in more detail in this section. Figure 12 shows the design for a ViewCube1km, the object which controls the display of a 1 km cube in the LViewer application. One or more ViewCube1km's would be attached to a ViewCube10km as the scene graph is built, shown at the very top of Figure 12.

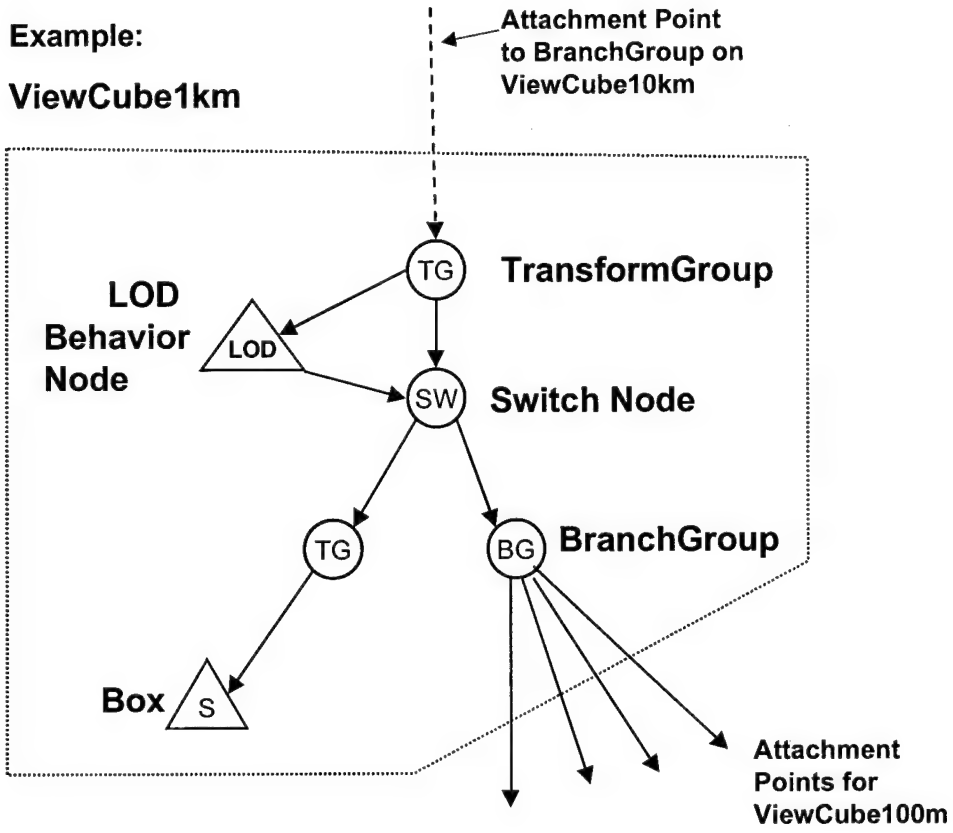

Figure 12. Illustration of the Inner workings of a typical ViewCube object.

ViewCube100m's would likewise be attached to the BranchGroup at the bottom of the figure, as needed.

Traversing the ViewCube structure shown in Figure 12, the first TransformGroup at the top serves as an attachment point for the Level of Detail (LOD) node below and to the left of it. The LOD node acts on the Switch node near the center. The function of the Level Of Detail object is to select one of the switch node's children based on distance from the viewer (Brown and Peterson, 1999). The most common application of the LOD object is to control the amount of detail, and therefore the amount of computing power, used in rendering an object in the scene.

For example, suppose the scene rendered by LViewer had a realistic model of the space shuttle sitting on a launch pad, an object consisting of a thousand points and hundreds of surfaces with complex shading and texturing. Each time the scene changes by even the slightest amount, each point must be transformed to its new, correct location, each surface redrawn and textured, all of which requires some amount of computing power. Now assume that the viewer is 40 km away, looking in the direction of the shuttle. The same extensive calculations for modeling the shuttle must still be performed, all for the sake of displaying a small white dot on the screen. A Level of Detail node can be used in this type of situation to keep the computational cost down. Beyond a given distance, perhaps 20 km or so, the shuttle might be represented as a white rectangle, since the viewer won't be able to make out any detail at this distance anyway. When the viewer moves within this cutoff distance, a somewhat more complex representation of the shuttle is displayed. Then, when the viewer moves within 5 km, the fully detailed model of the shuttle is used. As many levels of detail as desired can be used in this manner, to keep from using finite computational resources to display objects which are too far from the viewer to be seen clearly.

In the LViewer application, the LOD node is used to control the switching between different size cubes which is the core of the "Nested Cubes" visualization technique. The actual switching which takes place is between the two branches below the Switch node in Figure 12. The left branch contains the geometry for a 1 km transparent cube, along with the TransformGroup which places it in the proper position in the scene. The right branch holds a BranchGroup to which one or more ViewCube100m's are attached. During execution of the program, when the viewer is

outside of the cutoff distance contained in the LOD object, the left branch will be the active branch in the scene, and a 1 km cube will be visible. Once the viewer moves within the critical distance, the right branch becomes active, and the ViewCube100m objects attached to the BranchGroup become visible. The effect is one of large cubes dissolving into smaller cubes, and then individual points, as the viewer navigates closer.

Overall, four levels of detail are provided in the LViewer application. As shown in Figure 11, 10 km cubes give way to 1 km cubes, which turn into 100 m cubes, and finally dissolve into individual points. Each of the ViewCube objects in Figure 11 has the same structure as the ViewCube1km shown in Figure 12, with the exception of the 100 m cube, which has a list of points attached to its lower right BranchGroup instead of other cubes.

3.6.1.3 Texturing. Texturing is used in the LViewer application to give a more realistic appearance to the ground surface in the scene. The image used to create a texture in Java must have a height and width which are a power of 2 in order to be properly processed by the Java 3D API (Brown and Peterson, 1999). In this case, a 1024 X 1024 pixel map of the Cape Canaveral area is used to give the viewer visual cues as to where lightning is occurring. Using the Microsoft Streets98 program, a map was created which displayed the desirable level of detail, that could be cropped to 1024 X 1024 size. This modified image was referenced back to the original mapping program, and found to cover an area 14,800 m each direction from the LDAR central processing site, or an area 29,600 m square centered on the origin. A textured green border is placed around this map, so the total area of the ground surface in the scene is 40 km square. Lightning data is still displayed beyond this area, but without a surface map for reference. Java 3D will

not accept an image larger than 1024 pixels square for use in texturing, and while a larger, less detailed map might have been used, having an almost 30 km square map seemed adequate for displaying the significant areas of the Cape in sufficient detail.

Further attempts to create a more detailed map included using a 2048 X 2048 image (too large), and dividing the area into quads, each with a 1024 X 1024 image. While the latter method worked and gave a very detailed map, it slowed program performance considerably. MIP (Multum In Parvo, or many things in a small place) mapping was then employed in an attempt to get around the performance bottleneck when using the four quads. MIP mapping is a technique used to provide multiple levels of texture detail by creating several resolutions of the texture, each of which is derived by incrementally reducing the resolution of the original (Brown and Peterson, 1999). Performance was still noticeably slower, so the use of a higher resolution map was abandoned.

3.6.2 LViewer Execution. Having now examined some of the details involved in the inner workings of the LViewer program, this section will focus on how the application actually works from the perspective of the user.

Again, we may illustrate how a scene graph is built by taking the example of a file containing a single point, and following it through. Bear in mind that a file containing only one point would actually consist of 15 integers, or five sets of three coordinates: one set for each of the three cubes which contain the point, one for the point itself, and the last set as a flag for terminating the program. LViewer takes the intermediate data file constructed by LConvert and reads in the first line, or the first three integers. First, the range of the x coordinate is checked by the program to determine if the object is a

Cube10km, Cube1km, Cube100m, or a point (see section 3.5 for details on how the information is encoded). In every instance, the first object must be a Cube10km, so the application creates a ViewCube10km and places it in the Virtual Universe. The next three integers are read, the object is identified as a Cube1km, and a new ViewCube1km is created and attached to the ViewCube10km. The next object will be a Cube100m, so a ViewCube100m will be created and attached to the ViewCube1km. The last object to be read in from the file is a single point, which is attached to the ViewCube100m. A scene graph similar to Figure 11 has been constructed, but with only one of each object. The last three integers serve as an end-of-file tag for the program, which will exit when the z-coordinate is identified as a negative number.

The same procedure is followed for larger files, only a more complex scene graph is built. The application needs to have only one line at a time in memory to be able to build an accurate scene graph. After a given ViewCube10km is constructed, every object read in after that belongs within that cube, until either another Cube10km is encountered or the end of file is reached. Likewise, once a given ViewCube100m is created, each following point read in belongs within that cube until another cube is encountered or end of file is reached. This is equivalent to performing a leftmost traversal of the scene graph tree in Figure 11.

Once the LViewer application is running, the scene can be navigated using the left mouse button to spin the scene about its origin, the middle button to zoom, and the right button to move up/down and left/right. Additionally, the left/right arrow keys are used to spin the viewer/camera left or right, and the up/down arrow keys zoom in and out. Zooming with the arrow keys is much more rapid than using the mouse, so the arrow

keys are generally used to zoom in to the general area, then fine navigation can be performed with the mouse. Figures 13-18 show a sequence of screen captures for a typical LViewer display.

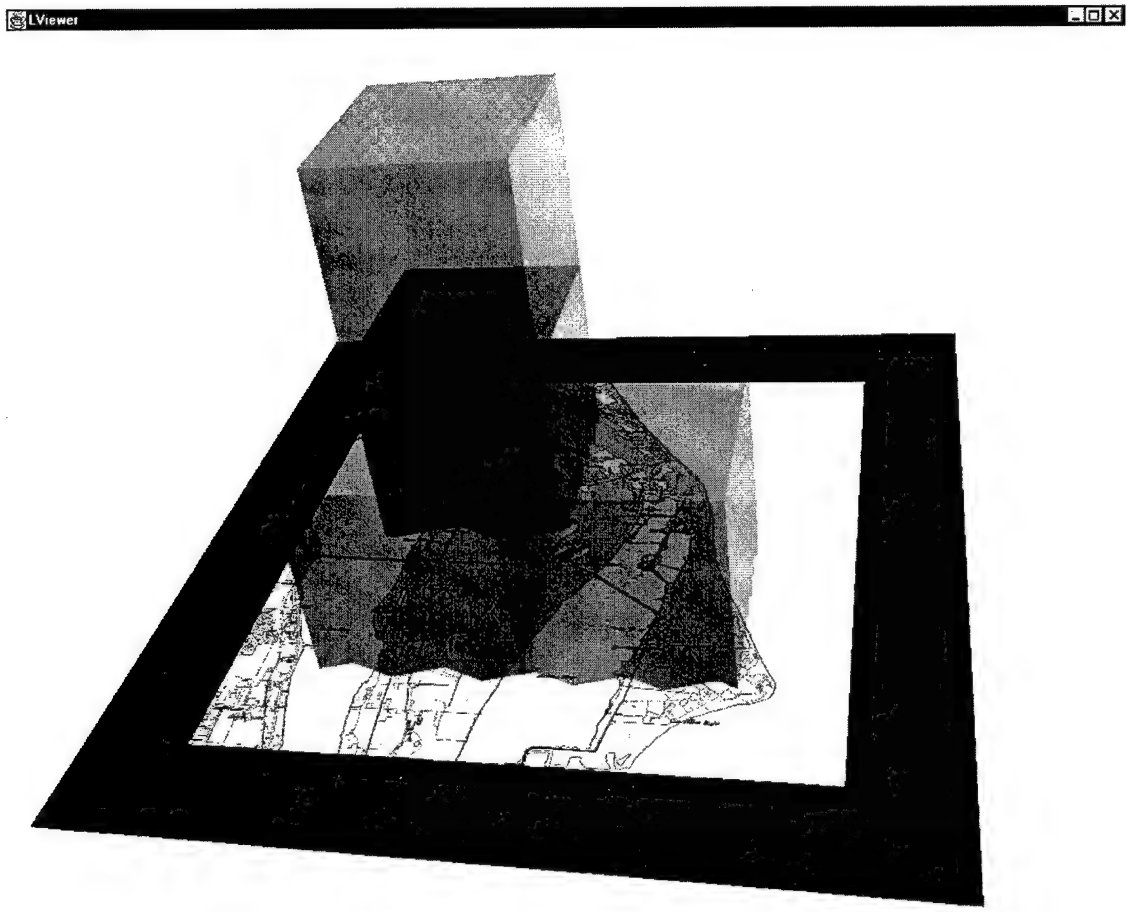

Figure 13. LViewer Overview of Cape Canaveral area. Ten km cubes show the general location of lightning activity in the scene.

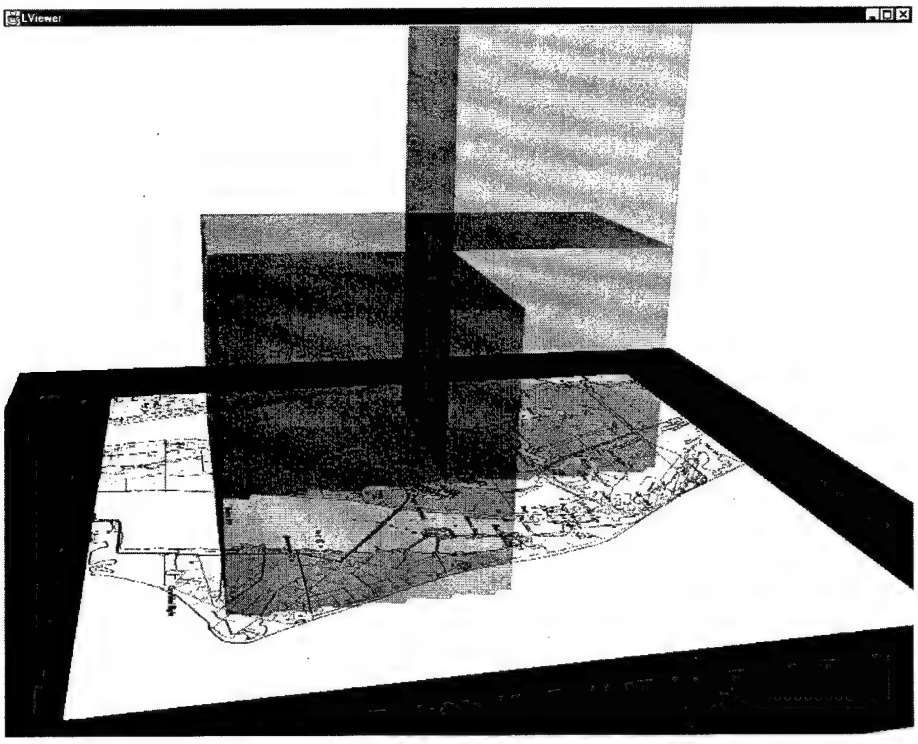

Figure 14. Same view as in Figure 13, rotated 90 degrees clockwise.

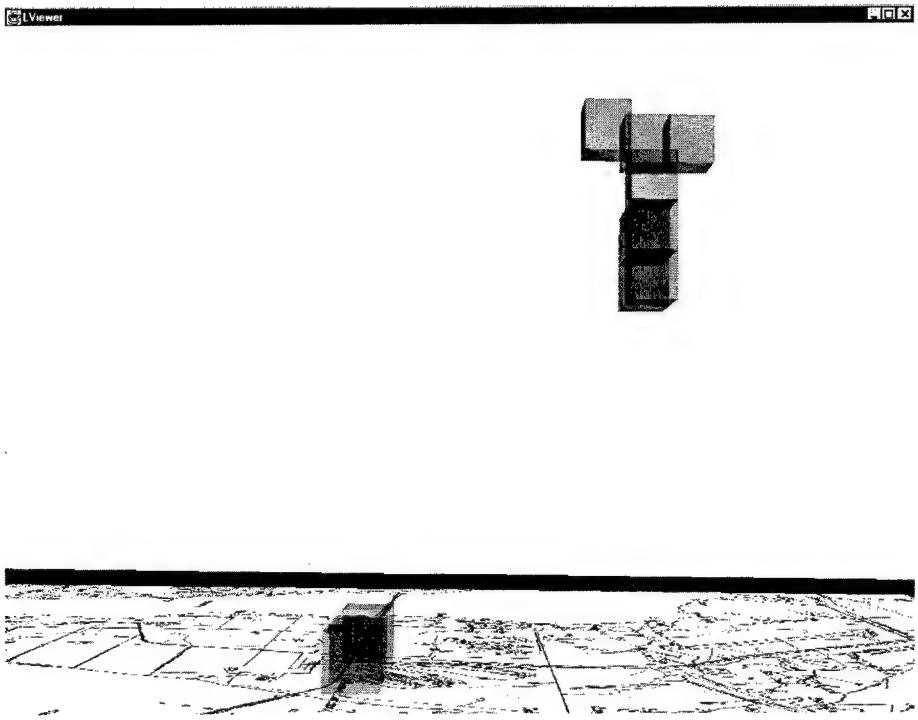

Figure 15. After zooming in toward the center of the map, the 10 km cubes in Figure 14 dissolve into several 1 km cubes.

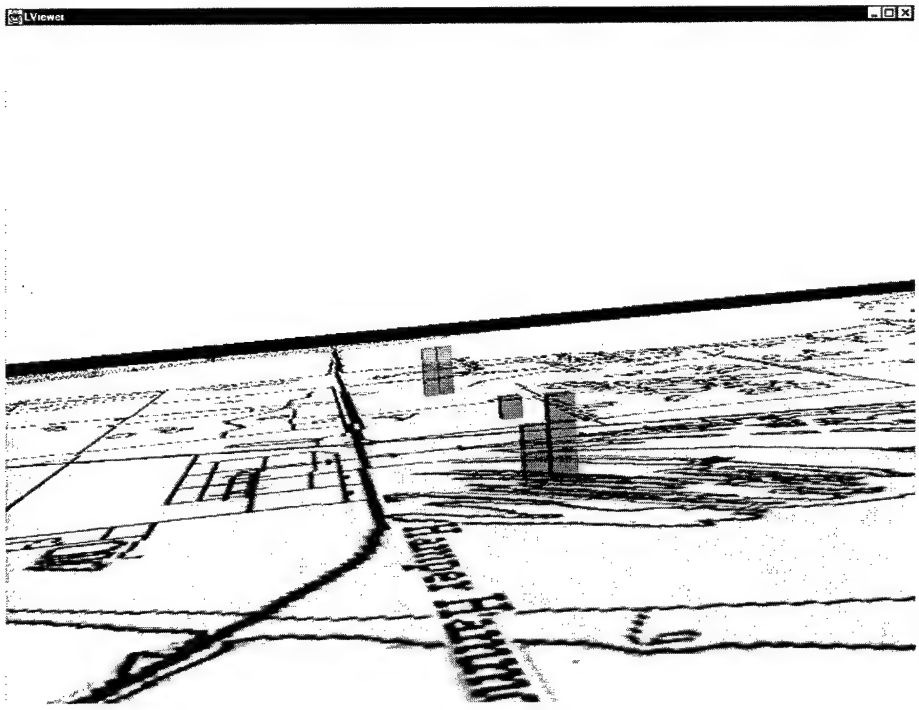

Figure 16. As the user zooms in on the 1 km cubes in the bottom left corner of Figure 15, they are replaced by 100 meter cubes.

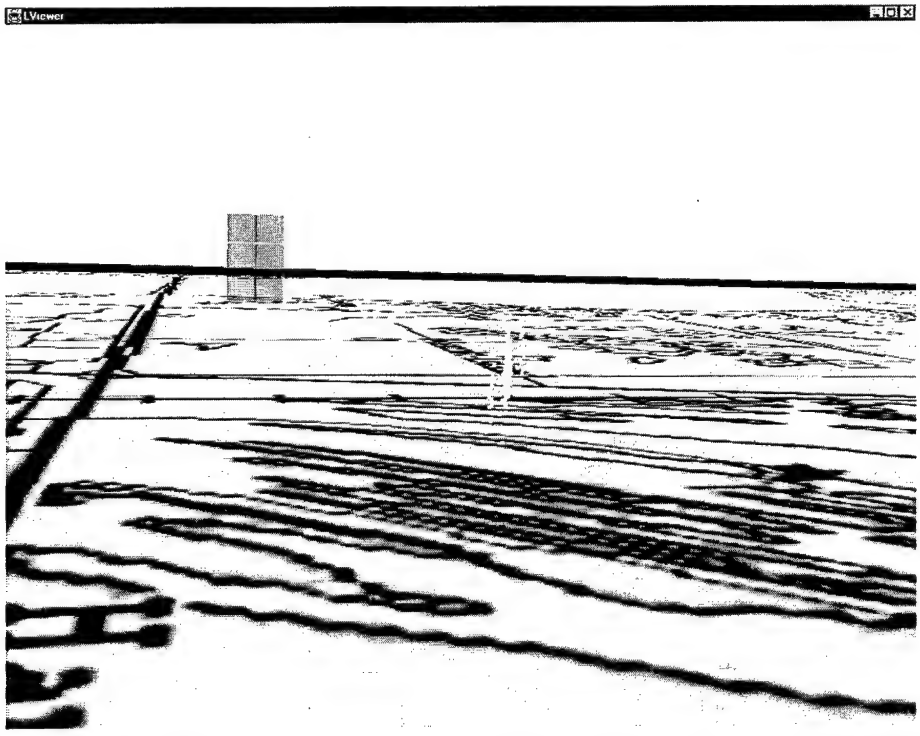

Figure 17. The cubes in the foreground of Figure 16 have dissolved into points, while the cubes in the background of Figure 16 remain unchanged.

Figure 18. The points in the foreground of Figure 17 are now behind the viewer, as the cubes in the background of Figure 17 dissolve into points.

4. Results and Conclusions

4.1 Overview

The previous chapters introduced and developed an application, LViewer, for the three-dimensional visualization of LDAR lightning data. LViewer successfully displays volumetric lightning data using a visualization technique called Nested Cubes designed to overcome the inherent problems involved in visualization of 3D data. A summary of the findings of this research and some recommendations for further work are discussed in the conclusion of this chapter.

4.2 Data

LDAR Data was retrieved from a web site run by the Global Hydrology Resource Center. The data was processed and written to an intermediate file in binary integer format for later ingestion by the display program. Duplicate data points were eliminated and extraneous information was discarded. This has the advantage of decreasing the overall size of the file by 50% to 90%, depending on the spatial distribution of the data in the file.

The size of the data sets involved proved to be a limiting factor, a common occurrence when dealing with large collections of environmental data. Files greater than about four megabytes could not be processed. Since the size of the data structures required to process a file is directly related to the number of unique data points, not the overall size of the file, this suggests that the critical factor is not the size of the file itself, but the dispersion of the data contained within. This was confirmed by testing a 16

megabyte file, consisting of a single two megabyte file repeated eight times, which could be processed successfully.

4.3 Display

The LViewer application was developed to demonstrate that the 3D visualization of volumetric lightning data is a viable concept, using the technique of Nested Cubes to minimize the problems inherent in displaying this type of data in three dimensions. LViewer loads in data from a file, constructs the appropriate Java 3D scene graph, and displays the lightning information in a three dimensional visualization which can be fully navigated by the user.

LViewer appears to be well-suited to the display of this particular type of data. Instead of treating the problem as a routine case of scientific visualization, the classic information visualization techniques of overview, zoom and filter, and details on demand are used to provide insight into the lightning phenomena. Beginning with an overview of the entire scene, the user can identify at a glance the areas with lightning activity, zoom in on areas of interest, and have the scene automatically change to display deeper levels of detail.

4.4 Summary

This research is intended to demonstrate the utility of using 3D visualization in the display of LDAR data. Using Java and Java 3D to develop the visualization application will allow it to be easily adapted to use over networks, including the Internet. As Air Force Weather moves toward more delivery of information over the web, the use of network-based technologies like Java is certain to become more prevalent.

Quantifying the effectiveness of a visualization system is not a straightforward matter, since it cannot be measured in the same way that rendering speeds or other more concrete metrics can (Botts, 1995). However, this application does demonstrate the utility of 3D visualization of LDAR data, as requested by the sponsor. Qualitative gains from this research include improved visualization of lightning not reaching the ground, which has potential use in both the aviation and scientific communities. As such, this thesis indicates directions for future research into lightning visualization and provides ideas to incorporate into the next generation LDAR display system.

4.5 Recommendations

Although the vast majority of LDAR data files can be viewed with LConvert and LViewer, there is still room for improvement in the area of processing large files. For this research, Java's powerful and easy to use data structures simplified the programming task, but a language with stronger numerical processing abilities such as Fortran or C would be more appropriate for converting the raw LDAR data into the intermediate file format. Considerable reduction in file size might also be realized by dividing and then multiplying each data point by 10 before processing. This would, in effect, give the application 10 m resolution instead of 1 m resolution in displaying individual data points. If one bears in mind that LDAR is a volumetric lightning warning system, not a system for counting the number of stepped leaders, then this reduction in resolution could be considered acceptable.

Another area for improvement would be the addition of time steps, to make the application 4-dimensional instead of 3-dimensional. However, as detailed in section

3.6.1.1, this would make data processing and scene graph construction considerably more complex. Additionally, the use of vector maps instead of textures to represent the ground surface in the scene is worth pursuing. Vector maps represent the outlines of coastline, roads, and other lines as a series of points which can be used to draw polygons representing features on the map. This would provide a more detailed ground surface without overtaxing the texturing process, as the LViewer application currently does, by pushing the upper limit of texture size (see section 3.6.1.3).

Probably the most interesting area for further investigation, from a research point of view, would be the integration of LDAR with radar data. A radar beam scans discrete volumes of space with each sweep, and assigns reflectivity values to that space. It is possible to display that volume of space graphically as a transparent object, similar to the cubes used in this research, along with the stepped leaders detected by LDAR. This combination of radar and LDAR could be used, among other things, to establish a link between reflectivity and probability of lightning, so that radar could be used as more of a predictive tool for lightning, rather than waiting for lightning to be detected by LDAR. Continued effort on these topics is encouraged.

Appendix A: Abbreviations

2D	Two-dimensional
3D	Three-dimensional
45 WS	45th Weather Squadron
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BG	BranchGroup
CCAS	Cape Canaveral Air Station
JIT	Just In Time
KSC	Kennedy Space Center
LDAR	Lightning Detection and Ranging
LOD	Level Of Detail
Mb	Megabyte
MIP	Multum In Parvo (many things in a small place)
NASA	National Aeronautics and Space Administration
OOP	Object-Oriented Programming
RF	Radio Frequency
TG	TransformGroup
TOA	Time of Arrival
US	United States
vtk	Visualization Toolkit

Appendix B: LViewer Program Code

```
/*
*     LViewer
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 05 Jan 00
*
*     Displays Lightning Detection and Ranging (LDAR) data in 3D.
*     Usage: java LViewer filename.dat, where filename.dat is an
*     LDAR data file which has been converted by the LConvert
*     program.  For larger files, the command java -mx64m LViewer
*     filename.dat may be used to increase the amount of memory
*     available to the program to 64 Mb.  The scene can be navigated
*     using the mouse and arrow keys.
*
*/

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.geometry.Box;
import com.sun.j3d.utils.geometry.Sphere;
import com.sun.j3d.utils.behaviors.mouse.*;
import com.sun.j3d.utils.behaviors.keyboard.*;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.image.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import java.util.*;
import java.io.*;


public class LViewer extends Applet
{

   // Creates the main BranchGroup which contains the appearance
   // and geometry for the scene graph
   public BranchGroup createSceneGraph(SimpleUniverse su, String
      nameOfFile)
   {
      // Define colors
      Color3f white = new Color3f(1.0f, 1.0f, 1.0f);
      Color3f black = new Color3f(0.0f, 0.0f, 0.0f);
      Color3f red   = new Color3f(0.80f, 0.20f, 0.2f);
      Color3f green = new Color3f(0.1f , 0.8f, 0.1f);
      Color3f ambientGreen = new Color3f(0.14f, 0.54f, 0.08f);
      Color3f ambientRed = new Color3f(0.2f, 0.05f, 0.0f);
      Color3f ambient = new Color3f(0.2f, 0.2f, 0.2f);
      Color3f diffuse = new Color3f(0.7f, 0.7f, 0.7f);
      Color3f specular = new Color3f(0.7f, 0.7f, 0.7f);
      // Define size of points
      float pointSize = 4.0f;
```

```
// Define scaling factor for entire scene
float scale = 0.01f;

// Create the red Appearance node
Material redMaterial = new Material(ambientRed, black, red,
   specular, 75.0f);
redMaterial.setLightingEnable(true);
Appearance redAppearance = new Appearance();
redAppearance.setMaterial(redMaterial);
ColoringAttributes redAttributes = new ColoringAttributes();
redAttributes.setColor(1.0f, 0.0f, 0.0f);
redAppearance.setColoringAttributes(redAttributes);
PointAttributes redPointAttributes = new
   PointAttributes(pointSize, false);
redAppearance.setPointAttributes(redPointAttributes);
TransparencyAttributes redTrans = new
   TransparencyAttributes(TransparencyAttributes.FASTEST, 0.5f);
redAppearance.setTransparencyAttributes(redTrans);

// Create the white Appearance node
Appearance whiteAppearance = new Appearance();
ColoringAttributes whiteAttributes = new ColoringAttributes();
whiteAttributes.setColor(1.0f, 1.0f, 1.0f);
whiteAppearance.setColoringAttributes(whiteAttributes);
PointAttributes whitePointAttributes = new
   PointAttributes(pointSize, false);
whiteAppearance.setPointAttributes(whitePointAttributes);

// Create the green Appearance object
Material greenMaterial =
new Material(ambientGreen, black, diffuse, specular, 75.0f);
greenMaterial.setLightingEnable(true);
Appearance greenAppearance = new Appearance();
greenAppearance.setMaterial(greenMaterial);

// Appearance for map
Appearance mapAppearance = new Appearance();

// Create the texture styles
TextureLoader[] tex;
tex = new TextureLoader[2];
tex[0] = new TextureLoader("grn128_2.jpg", this);
tex[1] = new TextureLoader("altCape4.jpg", this);

// Put textures onto the appropriate Appearance
greenAppearance.setTexture(tex[0].getTexture());
(greenAppearance.getTexture()).setBoundaryModeS(Texture.WRAP);
(greenAppearance.getTexture()).setBoundaryModeT(Texture.WRAP);
mapAppearance.setTexture(tex[1].getTexture());

// Create the root of the branch graph
BranchGroup objRoot = new BranchGroup();
```

```
// Attach lights to BranchGroup, so they don't rotate with scene
createLights(objRoot);


// Bounds for Mouse and other behaviors
BoundingSphere bounds =
   new BoundingSphere(new Point3d(0.0,0.0,0.0), 20000.0);

// Create a transform group node, initialized to the identity
// matrix.  Enable TRANSFORM_WRITE/READ so it can be modified
// at runtime.  Creat a Transform3D matrix to scale the scene
//
TransformGroup objTrans = new TransformGroup();
objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
objTrans.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
objRoot.addChild(objTrans);
Transform3D t3d = new Transform3D();
t3d.setScale(1.0);
t3d.rotY(1.58);
objTrans.setTransform(t3d);
BranchGroup branchRoot = new BranchGroup();
objTrans.addChild(branchRoot);

// Set up the background
Color3f bgColor = new Color3f(0.12f, 0.73f, 0.99f);
Background bgNode = new Background(bgColor);
bgNode.setApplicationBounds(bounds);
objRoot.addChild(bgNode);

// Open a file, read in data, and construct the SceneGraph
BranchGroup branch10km, branch1km, branch100m;
ViewCube10km currentVCube10km;
int cube10kmCount = -1;
int cube1kmCount, cube100mCount;
ViewCube1km currentVCube1km;
ViewCube100m currentVCube100m;
float[] verts;
Vector pointList;

// If no file name is passed by command line, use a default
// data file
if(nameOfFile.equals("default"))
{
   try
   {
      FileInputStream fin = new FileInputStream("Test.dat");
      System.out.println("----------" + "Test.dat" +
         "----------");
      DataInputStream din = new DataInputStream(fin);
      while (true)
      {
         int firstNumber = din.readInt();
         int secondNumber = din.readInt();
         int thirdNumber = din.readInt();
```

```
// A series of nested loops creates the set of nested
// cubes which ultimately contain a list of points at
// the end
while(firstNumber > 2500000)
{
   firstNumber -= 3000000;
   currentVCube10km = new ViewCube10km(firstNumber,
      secondNumber,thirdNumber,redAppearance);
   branchRoot.insertChild(currentVCube10km.getChild(),
      ++cube10kmCount);
   branch10km = currentVCube10km.getBranch();
   cube1kmCount = -1;

   firstNumber = din.readInt();
   secondNumber = din.readInt();
   thirdNumber = din.readInt();

   while(firstNumber < 2500000 && firstNumber > 1500000)
   {
      firstNumber -= 2000000;
      currentVCube1km = new ViewCube1km(firstNumber,
         secondNumber,thirdNumber,redAppearance);
      branch10km.insertChild(currentVCube1km.getChild(),
         ++cube1kmCount);
      branch1km = currentVCube1km.getBranch();
      cube100mCount = -1;

      firstNumber = din.readInt();
      secondNumber = din.readInt();
      thirdNumber = din.readInt();

      while(firstNumber < 1500000 && firstNumber >
         500000)
      {
         firstNumber -= 1000000;
         currentVCube100m = new ViewCube100m
            (firstNumber,secondNumber,thirdNumber,
            redAppearance);

         branch1km.insertChild(currentVCube100m.getChild
            (), ++cube100mCount);
         branch100m = currentVCube100m.getBranch();

         // Create vector (expandable array) for
         // temporarily holding points
         pointList = new Vector();
         Integer xPoint = new Integer(din.readInt());
         Integer yPoint = new Integer(din.readInt());
         Integer zPoint = new Integer(din.readInt());

         // Finally, compile list of points which fit
         // inside of 100m cube
         while(xPoint.intValue() < 500000 &&
            zPoint.intValue() >= 0)
         {
```

```
                        pointList.add(xPoint);
                        pointList.add(zPoint);
                        pointList.add(yPoint);
                        xPoint = new Integer(din.readInt());
                        yPoint = new Integer(din.readInt());
                        zPoint = new Integer(din.readInt());
                     }

                     // Now convert back to int from Integer
                     firstNumber = xPoint.intValue();
                     secondNumber = yPoint.intValue();
                     thirdNumber = zPoint.intValue();
                     verts = new float[pointList.size()];
                     for(int c = 0; c < pointList.size(); c++)
                     {
                        int temp = ((Integer)pointList.elementAt(c))
                           .intValue();
                        verts[c] = (float)temp*scale;
                     }

                     // PointArray holds individual lightning
                     // points, attaches to ViewCube100m
                     PointArray thesePoints = new PointArray(
                        (pointList.size()/3),
                        PointArray.COORDINATES);
                     thesePoints.setCoordinates(0, verts);
                     Shape3D lPoints = new Shape3D(thesePoints,
                        whiteAppearance);
                     branch100m.addChild(lPoints);
                  }
               }
            }
         }

         catch (EOFException e)
         {
            System.err.println(e);
         }

         catch (IOException e)
         {
            System.err.println(e);
         }
      }
      else
      {
         try
         {
            // Similar to nested loops above, but uses name of file
            // passed in from command line argument
            FileInputStream fin = new FileInputStream(nameOfFile +
               ".dat");
            System.out.println("----------" + nameOfFile +
               "----------");
```

```
DataInputStream din = new DataInputStream(fin);
while (true)
{
   int firstNumber = din.readInt();
   int secondNumber = din.readInt();
   int thirdNumber = din.readInt();

   while(firstNumber > 2500000)
   {
      firstNumber -= 3000000;
      currentVCube10km = new ViewCube10km(firstNumber,
         secondNumber,thirdNumber,redAppearance);
      branchRoot.insertChild(currentVCube10km.getChild(),
         ++cube10kmCount);
      branch10km = currentVCube10km.getBranch();
      cube1kmCount = -1;

      firstNumber = din.readInt();
      secondNumber = din.readInt();
      thirdNumber = din.readInt();

      while(firstNumber < 2500000 && firstNumber > 1500000)
      {
         firstNumber -= 2000000;
         currentVCube1km = new ViewCube1km(firstNumber,
            secondNumber,thirdNumber,redAppearance);
         branch10km.insertChild(currentVCube1km.getChild(),
            ++cube1kmCount);
         branch1km = currentVCube1km.getBranch();
         cube100mCount = -1;

         firstNumber = din.readInt();
         secondNumber = din.readInt();
         thirdNumber = din.readInt();

         while(firstNumber < 1500000 && firstNumber >
            500000)
         {
            firstNumber -= 1000000;
            currentVCube100m = new ViewCube100m
               (firstNumber,secondNumber,thirdNumber,
               redAppearance);

            branch1km.insertChild
               (currentVCube100m.getChild(),
               ++cube100mCount);
            branch100m = currentVCube100m.getBranch();

            // Create list for points
            pointList = new Vector();
            Integer xPoint = new Integer(din.readInt());
            Integer yPoint = new Integer(din.readInt());
            Integer zPoint = new Integer(din.readInt());
```

```
                    while(xPoint.intValue() < 500000 &&
                       zPoint.intValue() >= 0)
                    {
                       pointList.add(xPoint);
                       pointList.add(zPoint);
                       pointList.add(yPoint);
                       xPoint = new Integer(din.readInt());
                       yPoint = new Integer(din.readInt());
                       zPoint = new Integer(din.readInt());
                    }
                    // Now convert back to int
                    firstNumber = xPoint.intValue();
                    secondNumber = yPoint.intValue();
                    thirdNumber = zPoint.intValue();
                    verts = new float[pointList.size()];

                    for(int c = 0; c < pointList.size(); c++)
                    {
                       int temp = ((Integer)
                          pointList.elementAt(c)).intValue();
                       verts[c] = (float)temp*scale;
                    }

                    // PointArray holds individual lightning
                    // points, attaches to ViewCube100m
                    PointArray thesePoints = new PointArray(
                       (pointList.size()/3),
                       PointArray.COORDINATES);
                    thesePoints.setCoordinates(0, verts);
                    Shape3D lPoints = new Shape3D(thesePoints,
                       whiteAppearance);
                    branch100m.addChild(lPoints);
                 }
              }
           }
        }
     } // end else

     catch (EOFException e)
     {
        System.err.println(e);
     }
     catch (IOException e)
     {
        System.err.println(e);
     }

}


// Create the mouse drag behavior node
MouseRotate behavior = new MouseRotate();
behavior.setTransformGroup(objTrans);
objTrans.addChild(behavior);
behavior.setSchedulingBounds(bounds);
```

```
        // This sets the rate of spin
        behavior.setFactor(0.001);

        // Create the mouse zoom behavior node
        MouseZoom behavior2 = new MouseZoom();
        behavior2.setTransformGroup(objTrans);
        objTrans.addChild(behavior2);
        behavior2.setSchedulingBounds(bounds);

        // Create the mouse translate behavior node
        MouseTranslate behavior3 = new MouseTranslate();
        behavior3.setTransformGroup(objTrans);
        objTrans.addChild(behavior3);
        behavior3.setSchedulingBounds(bounds);

        // This sets the rate of X and Y translation
        behavior3.setFactor(0.04);

        // Create the arrow key navigation node, which can't
        // be adjusted for rate of speed
        TransformGroup suTrans = su.getViewingPlatform()
            .getViewPlatformTransform();
        KeyNavigatorBehavior keyNavBeh = new
            KeyNavigatorBehavior(suTrans);
        keyNavBeh.setSchedulingBounds(bounds);
        objRoot.addChild(keyNavBeh);

        // Add the ground surface
        objTrans.addChild(new Plane(greenAppearance).getChild());
        objTrans.addChild(new BigMap(mapAppearance).getChild());

        // Have Java 3D perform optimizations on this scene graph.
        objRoot.compile();

        return objRoot;

    }  // end createSceneGraph


    // Sets up lighting for scene
    private void createLights(BranchGroup graphRoot)
    {
        // Create a bounds for the light source influence
        BoundingSphere bounds =
            new BoundingSphere(new Point3d(0.0,0.0,0.0), 2000.0);
        Color3f white = new Color3f(1.0f, 1.0f, 1.0f);

        //Create the ambient light
        AmbientLight ambLight = new AmbientLight(white);
        ambLight.setInfluencingBounds(bounds);
        graphRoot.addChild(ambLight);

        //Create the directional light
        Vector3f dir = new Vector3f(-2.0f, -1.50f, -1.0f);
        DirectionalLight dirLight = new DirectionalLight(white, dir);
```

```
        dirLight.setInfluencingBounds(bounds);
        graphRoot.addChild(dirLight);

    } // end createLights


    // Brings together all of the elements needed to create
    // the scene
    public LViewer(String s)
    {
        setLayout(new BorderLayout());
        GraphicsConfiguration config =
            SimpleUniverse.getPreferredConfiguration();
        Canvas3D c = new Canvas3D(config);
        add("Center", c);

        // Create the scene and attach it to the virtual universe
        SimpleUniverse u = new SimpleUniverse(c);
        BranchGroup scene = createSceneGraph(u, s);

        // Set the ViewPlatform for an overview of the entire scene by
        // creating a temporary Transform3D object, setting it to the
        // current transform, performing rotation and translation on it,
        // then putting it back
        //
        Transform3D temp = new Transform3D();
        u.getViewingPlatform().getViewPlatformTransform()
            .getTransform(temp);
        Transform3D rot = new Transform3D();
        rot.rotX(-0.6);
        temp.mul(rot);
        Vector3d trans = new Vector3d(00.0, 360.0, 600.0);
        temp.setTranslation(trans);
        u.getViewingPlatform().getViewPlatformTransform()
            .setTransform(temp);

        // Set the back clipping distance so you can see the whole scene
        u.getViewer().getView().setBackClipDistance(1000.0);

        // Now add this SceneGraph onto the VirtualUniverse
        u.addBranchGraph(scene);

    } // end LViewer


    // The following allows this to be run as an application
    // as well as an applet, in a 1024 X 768 window
    //
    public static void main(String[] args)
    {
        if(!(args.length==0))
        {
            String fileName = args[0];
            new MainFrame(new LViewer(fileName), 1024, 768);
        }
```

```
        else
        {
            String fileName = new String("default");
            new MainFrame(new LViewer(fileName), 1024, 768);
        }

    } // end main


}  // end class LViewer
```

Appendix C: ViewCube10km Source Code

```
/*
*     ViewCube10km
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 29 Aug 99
*
*     Creates a 10 km cube for display in LViewer scene graph.
*
*/

import com.sun.j3d.utils.geometry.Box;
import com.sun.j3d.utils.geometry.Sphere;
import javax.media.j3d.*;
import javax.vecmath.*;

public class ViewCube10km
{
   private TransformGroup rootGroup;
   private BranchGroup cube1kmGroup;

   public ViewCube10km(int Xval, int Yval, int Zval, Appearance
      appearance)
   {
      float scale = 0.01f;
      double xVal, yVal, zVal;

      // Create a switch to hold the different levels of detail
      Switch sw = new Switch(0);
      sw.setCapability(javax.media.j3d.Switch.ALLOW_SWITCH_READ);
      sw.setCapability(javax.media.j3d.Switch.ALLOW_SWITCH_WRITE);

      // Create Groups to be used in this portion of SceneGraph
      rootGroup = new TransformGroup();
      cube1kmGroup = new BranchGroup();
      TransformGroup cubeGroup = new TransformGroup();

      // Calculate origin for cube; set TG to translate cube there
      if(Xval < 0) xVal = (double)(Xval + 5000);
      else         xVal = (double)(Xval - 5000);
      if(Yval < 0) yVal = (double)(Yval + 5000);
      else         yVal = (double)(Yval - 5000);
      zVal = (double)(Zval - 5000);
      Transform3D t3d = new Transform3D();
      // Note: next line converts from cartesian to graphics coords for
      // translation
      Vector3d trans = new Vector3d(xVal*scale, zVal*scale,
         yVal*scale);
      t3d.setTranslation(trans);
      cubeGroup.setTransform(t3d);
```

```
        // Add a BranchGroup to Switch which holds 1km cubes contained
        // w/in this cube
        sw.addChild(cube1kmGroup);
        // Add a TransformGroup to Switch which will translate cube
        sw.addChild(cubeGroup);
        // Add 10km cube onto this TG
        cubeGroup.addChild(new Box(5000.0f*scale,5000.0f*scale,
           5000.0f*scale,appearance));

        // Create bounds for LOD node, centered at origin of cube
        BoundingSphere bounds =
            new BoundingSphere(new Point3d(0.0,0.0,0.0), 3000.0);
        bounds.transform(t3d);

        // Set switching distance and origin for DistanceLOD behavior;
        // attach it to Switch
        float[] distances = new float[1];
        distances[0] = 300.0f;
        Point3f lodPoint = new Point3f((float)xVal*scale,
           (float)zVal*scale,(float)yVal*scale);
        DistanceLOD lod = new DistanceLOD(distances, lodPoint);
        lod.setSchedulingBounds(bounds);
        lod.addSwitch(sw);

        // Add LOD behavior node and Switch to top TransformGroup
        rootGroup.addChild(lod);
        rootGroup.addChild(sw);
    }


    // Allows instance of this class to be attached to SceneGraph via TG
    public TransformGroup getChild()
    {
        return rootGroup;
    }


    // Returns BG allowing smaller cubes in the SceneGraph to be
    // attached
    public BranchGroup getBranch()
    {
        return cube1kmGroup;
    }


}  // end class ViewCube10km
```

Appendix D: ViewCube1km Source Code

```
/*
*     ViewCube1km
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 29 Aug 99
*
*     Creates a 1 km cube for display in LViewer scene graph.
*
*/

import com.sun.j3d.utils.geometry.Box;
import com.sun.j3d.utils.geometry.Sphere;
import javax.media.j3d.*;
import javax.vecmath.*;

public class ViewCube1km
{
   private TransformGroup rootGroup;
   private BranchGroup cube100mGroup;

   public ViewCube1km(int Xval, int Yval, int Zval, Appearance
      appearance)
   {
      float scale = 0.01f;
      double xVal, yVal, zVal;

      // Create a switch to hold the different levels of detail
      Switch sw = new Switch(0);
      sw.setCapability(javax.media.j3d.Switch.ALLOW_SWITCH_READ);
      sw.setCapability(javax.media.j3d.Switch.ALLOW_SWITCH_WRITE);

      // Create Groups to be used in this portion of SceneGraph
      rootGroup = new TransformGroup();
      cube100mGroup = new BranchGroup();
      TransformGroup cubeGroup = new TransformGroup();

      // Create a TG and add it to the switch
      if(Xval < 0) xVal = (double)(Xval + 500);
      else         xVal = (double)(Xval - 500);
      if(Yval < 0) yVal = (double)(Yval + 500);
      else         yVal = (double)(Yval - 500);
      zVal = (double)(Zval - 500);
      Transform3D t3d = new Transform3D();
      // Note: next line converts from cartesian to graphics coords for
      // translation
      Vector3d trans = new Vector3d(xVal*scale, zVal*scale,
         yVal*scale);
      t3d.setTranslation(trans);
      cubeGroup.setTransform(t3d);
```

```
    // Add a BranchGroup to Switch which holds 100m cubes contained
    // w/in this cube
    sw.addChild(cube100mGroup);
    // Add a TransformGroup to Switch which will translate cube
    sw.addChild(cubeGroup);
    // Add 1km cube onto this TG
    cubeGroup.addChild(new Box(500.0f*scale, 500.0f*scale,
       500.0f*scale, appearance));

    // Create bounds for LOD node, centered at origin of cube
    BoundingSphere bounds =
        new BoundingSphere(new Point3d(0.0,0.0,0.0), 100000.0);
    bounds.transform(t3d);

    // Set switching distance and origin for DistanceLOD behavior
    float[] distances = new float[1];
    distances[0] = 100.0f;
    Point3f lodPoint = new Point3f((float)xVal*scale,
       (float)zVal*scale,(float)yVal*scale);
    DistanceLOD lod = new DistanceLOD(distances, lodPoint);
    lod.setSchedulingBounds(bounds);
    lod.addSwitch(sw);

    // Add LOD behavior node and Switch to top TransformGroup
    rootGroup.addChild(lod);
    rootGroup.addChild(sw);
}


// Allows instance of this class to be attached to SceneGraph via TG
public TransformGroup getChild()
{
    return rootGroup;
}


// Returns BG allowing smaller cubes in the SceneGraph to be
// attached
public BranchGroup getBranch()
{
    return cube100mGroup;
}


    }  // end class ViewCube1km
```

Appendix E: ViewCube100m Source Code

```
/*
*      ViewCube100m
*
*      Developed by
*      Lt Michael W. Darwin, USAF
*      Last Modified: 08 Nov 99
*
*      Creates a 100 m cube for display in LViewer scene graph.
*
*/

import com.sun.j3d.utils.geometry.Box;
import com.sun.j3d.utils.geometry.Sphere;
import javax.media.j3d.*;
import javax.vecmath.*;

public class ViewCube100m
{
   private TransformGroup rootGroup;
   private BranchGroup pointGroup;

   public ViewCube100m(int Xval, int Yval, int Zval, Appearance
      appearance)
   {
      float scale = 0.01f;
      double xVal, yVal, zVal;

      // Create a switch to hold the different levels of detail
      Switch sw = new Switch(0);
      sw.setCapability(javax.media.j3d.Switch.ALLOW_SWITCH_READ);
      sw.setCapability(javax.media.j3d.Switch.ALLOW_SWITCH_WRITE);

      // Create Groups to be used in this portion of SceneGraph
      rootGroup = new TransformGroup();
      pointGroup = new BranchGroup();
      TransformGroup cubeGroup = new TransformGroup();

      // Calculate origin for cube; set TG to translate cube there
      if(Xval < 0) xVal = (double)(Xval + 50);
      else         xVal = (double)(Xval - 50);
      if(Yval < 0) yVal = (double)(Yval + 50);
      else         yVal = (double)(Yval - 50);
      zVal = (double)(Zval - 50);
      Transform3D t3d = new Transform3D();
      // Note: next line converts from cartesian to graphics coords for
      // translation
      Vector3d trans = new Vector3d(xVal*scale, zVal*scale,
         yVal*scale);
      t3d.setTranslation(trans);
      cubeGroup.setTransform(t3d);
```

```
        // Add a BranchGroup to Switch which holds points contained w/in
        // this cube
        sw.addChild(pointGroup);
        // Add a TransformGroup to Switch which will translate cube
        sw.addChild(cubeGroup);
        // Add 100m cube onto this TG
        cubeGroup.addChild(new Box(47.5f*scale,47.5f*scale,47.5f*scale,
           appearance));

        // Create bounds for LOD node, centered at origin of cube
        BoundingSphere bounds =
            new BoundingSphere(new Point3d(0.0,0.0,0.0), 10000.0);
        bounds.transform(t3d);

        // Set switching distance and origin for DistanceLOD behavior;
        // attach it to Switch
        float[] distances = new float[1];
        distances[0] = 40.0f;
        Point3f lodPoint = new Point3f((float)xVal*scale,
           (float)zVal*scale,(float)yVal*scale);
        DistanceLOD lod = new DistanceLOD(distances, lodPoint);
        lod.setSchedulingBounds(bounds);
        lod.addSwitch(sw);

        // Add LOD behavior node and Switch to top TransformGroup
        rootGroup.addChild(lod);
        rootGroup.addChild(sw);
    }


    // Allows instance of this class to be attached to SceneGraph via TG
    public TransformGroup getChild()
    {
        return rootGroup;
    }


    // Returns BG allowing smaller cubes in the SceneGraph to be
    // attached
    public BranchGroup getBranch()
    {
        return pointGroup;
    }


}  // end class ViewCube100m
```

Appendix F: Plane Source Code

```
/*
 *    @(#)Plane.java 1.3 98/02/20 14:30:08
 *
 * Copyright (c) 1996-1998 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license
 * to use, modify and redistribute this software in source and binary
 * code form, provided that i) this copyright notice and license appear
 * on all copies of the software; and ii) Licensee does not utilize the
 * software in a manner which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind.
 * ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES,
 * INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND
 * ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY
 * LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE
 * SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS
 * BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES,
 * HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING
 * OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control
 * of aircraft, air traffic, aircraft navigation or aircraft
 * communications; or in the design, construction, operation or
 * maintenance of any nuclear facility. Licensee represents and
 * warrants that it will not use or redistribute the Software for such
 * purposes.
 *
 *    Modified by
 *    Lt Michael W. Darwin, USAF
 *    Last Modified: 30 Nov 99
 *
 *    Creates a textured border around a 296 X 296 square centered on
 *    the origin
 *
 */

import java.applet.Applet;
import java.awt.event.*;
import javax.media.j3d.*;
import javax.vecmath.*;

public class Plane extends Object
{

   private Shape3D shape3D;
```

```
    private static final float[] verts =
    {
       // Ground surface
       -200.0f, -0.001f, 200.0f,   148.0f, -0.001f, 200.0f,
       148.0f, -0.001f, 148.0f,   -200.0f, -0.001f, 148.0f,

       148.0f, -0.001f, 200.0f,   200.0f, -0.001f, 200.0f,
       200.0f, -0.001f, -148.0f,  148.0f, -0.001f, -148.0f,

       -148.0f, -0.001f, -148.0f,   200.0f, -0.001f, -148.0f,
       200.0f, -0.001f, -200.0f,   -148.0f, -0.001f, -200.0f,

       -200.0f, -0.001f, 148.0f,   -148.0f, -0.001f, 148.0f,
       -148.0f, -0.001f, -200.0f,   -200.0f, -0.001f, -200.0f,

    };

    private static final float[] normals =
    {
       // Ground Surface
       0.0f, 1.0f,  0.0f,     0.0f, 1.0f,  0.0f,
       0.0f, 1.0f,  0.0f,     0.0f, 1.0f,  0.0f,
    };

    private Point2f texCoord[] =
    {
       new Point2f(5.0f, 0.0f), new Point2f(5.0f, 5.0f),
       new Point2f(0.0f, 5.0f), new Point2f(0.0f, 0.0f)
    };

    public Plane(Appearance appearance)
    {
       QuadArray quadArray = new QuadArray(16, QuadArray.COORDINATES |
                                 QuadArray.NORMALS |
                                 QuadArray.TEXTURE_COORDINATE_2);
       quadArray.setCoordinates(0, verts);
       quadArray.setNormals(0, normals);

       for (int i = 0; i < 16; i++)
       {
          quadArray.setTextureCoordinate(i, texCoord[i%4]);
       }

       shape3D = new Shape3D(quadArray, appearance);
    }

    public Shape3D getChild()
    {
       return shape3D;
    }

}
```

Appendix G: BigMap Source Code

```
/*
 *    BigMap.java modified from:
 *    @(#)Plane.java 1.3 98/02/20 14:30:08
 *
 * Copyright (c) 1996-1998 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license
 * to use, modify and redistribute this software in source and binary
 * code form, provided that i) this copyright notice and license appear
 * on all copies of the software; and ii) Licensee does not utilize the
 * software in a manner which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind.
 * ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES,
 * INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND
 * ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY
 * LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE
 * SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS
 * BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES,
 * HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING
 * OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control
 * of aircraft, air traffic, aircraft navigation or aircraft
 * communications; or in the design, construction, operation or
 * maintenance of any nuclear facility. Licensee represents and
 * warrants that it will not use or redistribute the Software for such
 * purposes.
 *
 *    Modified by
 *    Lt Michael W. Darwin, USAF
 *    Last Modified: 23 Nov 99
 *
 *    Creates a textured 296 X 296 square centered on the origin
 *
 */

import java.applet.Applet;
import java.awt.event.*;
import javax.media.j3d.*;
import javax.vecmath.*;

public class BigMap extends Object
{

   private Shape3D shape3D;
```

```
    private static final float[] verts =
    {
       // Ground surface
       -148.0f, -0.001f,  148.0f,   148.0f, -0.001f, 148.0f,
       148.0f, -0.001f, -148.0f,   -148.0f, -0.001f,  -148.0f,
    };

    private static final float[] normals =
    {
       // Ground Surface
       0.0f, 1.0f,  0.0f,     0.0f, 1.0f,  0.0f,
       0.0f, 1.0f,  0.0f,     0.0f, 1.0f,  0.0f,
    };

    private static final float[] textCoords =
    {
       // Ground Surface
       1.0f,  0.0f,            1.0f,  1.0f,
       0.0f,  1.0f,            0.0f,  0.0f
    };

    public BigMap(Appearance appearance)
    {
       QuadArray quadArray = new QuadArray(4, QuadArray.COORDINATES |
                                 QuadArray.NORMALS |
                                 QuadArray.TEXTURE_COORDINATE_2);
       quadArray.setCoordinates(0, verts);
       quadArray.setNormals(0, normals);
       quadArray.setTextureCoordinates(0, textCoords);

       shape3D = new Shape3D(quadArray, appearance);
    }

    public Shape3D getChild()
    {
       return shape3D;
    }

}
```

Appendix H: LConvert Source Code

```
/*
*     LConvert
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 29 Nov 99
*
*     Uses instance of LDataConverter to convert a raw
*     LDAR file (*.asc) into an intermediate data file
*     (*.dat)
*
*/

import java.util.*;
import java.io.*;


public class LConvert
{

   public static void main(String[] args)
   {
      // LDataConverter object reads in LDAR data and puts it in data
      // structures.  Open a default file if no file argument is passed
      // when program is run
      //
      LDataConverter myTest;
      String fileName;

      if(!(args.length==0))
      {
         fileName = args[0];
      }
      else
      {
         fileName = "990402_testme";
      }

      myTest = new LDataConverter(fileName);
      // Uncomment following line to write processed data file to
      // screen
      //myTest.listContents();
      // Writes output to file
      myTest.writeContents(fileName);
   }

}  // end class LConvert
```

Appendix I: LDataConverter Source Code

```
/*
*     LDataConverter
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified:  29 Nov 99
*
*     Uses instances of TheCube, Cube10km, Cube1km,
*     Cube100m (versions of a HashTable), and MyPoints
*     to arrange data points from a raw LDAR file into
*     a treelike data structure, then perform a leftmost
*     traversal of tree to write output to file or screen
*
*/

import java.io.*;
import java.util.*;

public class LDataConverter
{
   public TheCube theCube = new TheCube();
   Integer temp;

   // Constructor
   public LDataConverter(String s)
   {
      try
      {
         // Create a File object and an input stream object for the
         // file
         String inFileName = "LDAR_Packets_"+s+".asc"; // File name
         File myLData = new File(inFileName);
         FileReader myFileReader = new FileReader(myLData);

         // StreamTokenizer gets tokens from input stream
         StreamTokenizer myStream = new StreamTokenizer(myFileReader);

         // Array for reading one line of ints
         int[] val = new int[3];

         try
         {
            while(myStream.nextToken() != StreamTokenizer.TT_EOF)
            {
               int index = 0;  // Index for storing ints

               for(index = 0; index < 5; index++)
               {
                  myStream.nextToken();  // Skip first 5 integers
               }
               // Put x, y, and z values into array
               val[0] = (int)myStream.nval;
               myStream.nextToken();
```

```
            val[1] = (int)myStream.nval;
            myStream.nextToken();
            val[2] = (int)myStream.nval;

            // First, see if the proper Cube10km already exists by
            // calculating its hashcode and searching for it; if it
            // doesn't exist, create it and put it in HashMap
            // "TheCube"
            temp = new Integer
               (Cube10km.calcKey(val[0],val[1],val[2]));
            if(!theCube.containsKey(temp))
               {
                  theCube.put(temp, new Cube10km
                     (val[0],val[1],val[2]));
               }

            // Now the proper 10km cube exists, so get its handle,
            // and repeat above procedure with 1km cubes
            Cube10km map10 = (Cube10km)theCube.get(temp);
            temp = new Integer
               (Cube1km.calcKey(val[0],val[1],val[2]));
            if(!map10.containsKey(temp))
               {
                  map10.put(temp, new Cube1km
                     (val[0],val[1],val[2]));
               }

            // Now the proper 1km cube exists, so get its handle,
            // and repeat above procedure with 100m cubes
            Cube1km map1 = (Cube1km)map10.get(temp);
            temp = new Integer(Cube100m.calcKey
               (val[0],val[1],val[2]));
            if(!map1.containsKey(temp))
               {
                  map1.put(temp, new Cube100m
                     (val[0],val[1],val[2]));
               }

            // Finally, the proper 100m cube exists, so add new set
            // of points to HashSet
            ((Cube100m)map1.get(temp)).add(new MyPoints
               (val[0],val[1],val[2]));

         } // end while

      } // inner try block

      catch(IOException e)
      {
         System.out.println("Error reading input stream; " + e );
      }

   // Close the input stream
      myFileReader.close();
```

```
    } // end main try block

    // Stream creation exception
    catch(FileNotFoundException e)
    {
       System.err.println("File not found in LDataLoader; "  + e);
       return;
    }

    // File read exception
    catch(IOException e)
    {
       System.out.println("IO Error reading input file; " + e );
       return;
    }

} // end constructor


// Method to move through the data structure and list all objects it
// contains to screen
void listContents()
{
    // Collection container to hold HashMap values
    Collection theCubeVals = theCube.values();
    // Iterator over the collection
    Iterator theCubeIt = theCubeVals.iterator();
    // Move through all the 10km cubes in theCube
    while(theCubeIt.hasNext())
    {
       Cube10km map10 = (Cube10km)theCubeIt.next();
       map10.printVal();

       // Collection container to hold HashMap values
       Collection map10Vals = map10.values();
       // Iterator over the collection
       Iterator Cube10kmIt = map10Vals.iterator();
       // Move through all the 1km cubes in Cube10km
       while(Cube10kmIt.hasNext())
       {
          Cube1km map1 = (Cube1km)Cube10kmIt.next();
          map1.printVal();

          // Collection container to hold HashMap values
          Collection map1Vals = map1.values();
          // Iterator over the collection
          Iterator Cube1kmIt = map1Vals.iterator();
          // Move through all the 100m cubes in Cube1km
          while(Cube1kmIt.hasNext())
          {
             // a is handle to next Cube100m object
             Cube100m a = (Cube100m)Cube1kmIt.next();
             // Print it to make sure you "see" it
             a.printVal();
```

```
                // create an iterator for HashSet containing points
                Iterator Cube100mIt = a.iterator();
                while(Cube100mIt.hasNext())
                {
                    MyPoints b = (MyPoints)Cube100mIt.next();
                    // Print out point values
                    b.printVal();
                }
            }
        }
    }
}       // end listContents


// Method to move through the data structure and write all objects
// it contains to output file
void writeContents(String s)
{
    DataOutputStream dos = null;
    // Collection container to hold HashMap values
    Collection theCubeVals = theCube.values();
    // Iterator over the collection
    Iterator theCubeIt = theCubeVals.iterator();

    try
    {
        dos = new DataOutputStream(new FileOutputStream(s + ".dat"));

        // Move through all the 10km cubes in theCube
        while(theCubeIt.hasNext())
        {
            Cube10km map10 = (Cube10km)theCubeIt.next();
            dos.writeInt(map10.getXval()+3000000);
            dos.writeInt(map10.getYval());
            dos.writeInt(map10.getZval());

            // Collection container to hold HashMap values
            Collection map10Vals = map10.values();
            // Iterator over the collection
            Iterator Cube10kmIt = map10Vals.iterator();
            // Move through all the 1km cubes in Cube10km
            while(Cube10kmIt.hasNext())
            {
                Cube1km map1 = (Cube1km)Cube10kmIt.next();
                dos.writeInt(map1.getXval()+2000000);
                dos.writeInt(map1.getYval());
                dos.writeInt(map1.getZval());

                // Collection container to hold HashMap values
                Collection map1Vals = map1.values();
                // Iterator over the collection
                Iterator Cube1kmIt = map1Vals.iterator();
                // Move through all the 100m cubes in Cube1km
                while(Cube1kmIt.hasNext())
                {
```

```
                    // a is handle to next Cube100m object
                    Cube100m a = (Cube100m)Cube1kmIt.next();
                    dos.writeInt(a.getXval()+1000000);
                    dos.writeInt(a.getYval());
                    dos.writeInt(a.getZval());

                    // create an iterator for HashSet containing points
                    Iterator Cube100mIt = a.iterator();
                    while(Cube100mIt.hasNext())
                    {
                       // Write point values to output file
                       MyPoints b = (MyPoints)Cube100mIt.next();
                       dos.writeInt(b.getXval());
                       dos.writeInt(b.getYval());
                       dos.writeInt(b.getZval());
                    }
                 }
              }
           }

           // Write flag to indicate end-of-file
           dos.writeInt(1);
           dos.writeInt(1);
           dos.writeInt(-1);

        }     // end try

        catch (IOException e) {System.err.println(e);}

        finally
        {
           try { if (dos != null) dos.close(); }
           catch (IOException e){}
        }
     }        // end writeContents


     // Return an iterator for "theCube", which is at the root of the
     // data structure
     public Iterator getIterator()
     {
        Collection theCubeVals = theCube.values();
        Iterator theCubeIt = theCubeVals.iterator();
        return theCubeIt;
     }

}    // end class LDataConverter
```

Appendix J: TheCube Source Code

```
/*
*     TheCube
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 29 Aug 99
*
*     Creates a HashMap for use by LConvert application
*     to hold Cube10km objects
*
*/

import java.util.*;
import javax.vecmath.Point3d;

public class TheCube extends HashMap
{
   public Map myMap;

   // Constructor
   public TheCube()
   {
      myMap = new HashMap();
   }

   void printVal()
   {
      System.out.println("HashMap 'TheCube' created");
   }

   public boolean equals(Object aName)
   {
      if(!this.getClass().getName().equals(aName.getClass().getName()))
         return false;
      else
         return true;
   }

} // end TheCube
```

Appendix K: Cube10km Source Code

```
/*
*     Cube10km
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 01 Nov 99
*
*     Creates a HashMap for use by LConvert application
*     to hold Cube1km objects.  Instances of this class
*     are inserted into TheCube object.
*
*/

import java.util.*;

public class Cube10km extends HashMap
{
   private int Xval;
   private int Yval;
   private int Zval;
   private int xval, yval, zval;
   public Map myMap;

   // Constructor
   public Cube10km(int Xval, int Yval, int Zval)
   {
      this.Xval = Xval;
      this.Yval = Yval;
      this.Zval = Zval;

      // Use div and mod to round x, y, and z
      // to the nearest 10000; this is all we
      // need to calculate hashcode for a given
      // 10km cube
      //
      if(Xval < 0)
         xval = (Xval/10000*10000 - 10000);
      else
         xval = (Xval/10000*10000 + 10000);
      if(Yval < 0)
         yval = (Yval/10000*10000 - 10000);
      else
         yval = (Yval/10000*10000 + 10000);

      zval = (Zval/10000*10000 + 10000);
      myMap = new HashMap();
   }
```

```
public Cube10km()
{
   this.Xval = 0;
   this.Yval = 0;
   this.Zval = 0;
   myMap = new HashMap();
}

public int getXval()
{
   return this.xval;
}

public int getYval()
{
   return this.yval;
}

public int getZval()
{
   return this.zval;
}

// Overrides hashCode() method for class Object
public int hashCode()
{
   return (xval/10000*7 + yval/10000*13 + zval/10000*19);
}

// Calculate hashcode for given x, y, z values; this allows a
// hashcode to be calculated without having to actually create
// a Cube10km object
//
public static int calcKey(int X, int Y, int Z)
{
   int tempX, tempY, tempZ;

   if(X < 0)
      tempX = (X/10000*10000 - 10000);
   else
      tempX = (X/10000*10000 + 10000);
   if(Y < 0)
      tempY = (Y/10000*10000 - 10000);
   else
      tempY = (Y/10000*10000 + 10000);

   tempZ = (Z/10000*10000 + 10000);

   return(tempX/10000*7 + tempY/10000*13 + tempZ/10000*19);
}
```

```
    void printVal()
    {
       System.out.println("   10km Cube at "+this.xval+" "+this.yval+"
          "+this.zval);
    }
    // Overrides equals() method for class Object
    public boolean equals(Object aName)
    {
       if(!this.getClass().getName().equals(aName.getClass().getName()))
          return false;
       else
          return (this.hashCode()==((Cube1km)aName).hashCode());
    }

} // end Cube10km
```

Appendix L: Cube1km Source Code

```
/*
*     Cube1km
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 30 Aug 99
*
*     Creates a HashMap for use by LConvert application
*     to hold Cube100m objects.  Instances of this class
*     are inserted into Cube10km object.
*
*/

import java.util.*;

public class Cube1km extends HashMap
{
   private int Xval;
   private int Yval;
   private int Zval;
   private int xval, yval, zval;
   public Map myMap;

   // Constructor
   public Cube1km(int Xval, int Yval, int Zval)
   {
      this.Xval = Xval;
      this.Yval = Yval;
      this.Zval = Zval;

      // Use div and mod to round x, y, and z
      // to the nearest 1000; this is all we
      // need to calculate hashcode for a given
      // 1km cube
      //
      if(Xval < 0)
         xval = (Xval/1000*1000 - 1000);
      else
         xval = (Xval/1000*1000 + 1000);
      if(Yval < 0)
         yval = (Yval/1000*1000 - 1000);
      else
         yval = (Yval/1000*1000 + 1000);

      zval = (Zval/1000*1000 + 1000);
      myMap = new HashMap();
   }
```

```
public Cube1km()
{
   this.Xval = 0;
   this.Yval = 0;
   this.Zval = 0;
   myMap = new HashMap();
}


public int getXval()
{
   return this.xval;
}

public int getYval()
{
   return this.yval;
}

public int getZval()
{
   return this.zval;
}


// Overrides hashCode() method for class Object
public int hashCode()
{
   return (xval/1000*7 + yval/1000*13 + zval/1000*19);
}


// Calculate hashcode for given x, y, z values; this allows a
// hashcode to be calculated without having to actually create
// a Cube1km object
//
public static int calcKey(int X, int Y, int Z)
{
   int tempX, tempY, tempZ;

   if(X < 0)
      tempX = (X/1000*1000 - 1000);
   else
      tempX = (X/1000*1000 + 1000);
   if(Y < 0)
      tempY = (Y/1000*1000 - 1000);
   else
      tempY = (Y/1000*1000 + 1000);

   tempZ = (Z/1000*1000 + 1000);

   return(tempX/1000*7 + tempY/1000*13 + tempZ/1000*19);
}
```

```
    void printVal()
    {
       System.out.println("   1km Cube at "+this.xval+" "+this.yval+"
       "+this.zval);
    }


    // Overrides equals() method for class Object
    public boolean equals(Object aName)
    {
       if(!this.getClass().getName().equals(aName.getClass().getName()))
          return false;
       else
          return (this.hashCode()==((Cube1km)aName).hashCode());
    }

} // end Cube1km
```

Appendix M: Cube100m Source Code

```
/*
*     Cube100m
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 30 Aug 99
*
*     Creates a HashSet for use by LConvert application
*     to hold MyPoints objects.  Instances of this class
*     are inserted into Cube1km object.
*
*/

import java.util.*;

public class Cube100m extends HashSet
{
   private int Xval;
   private int Yval;
   private int Zval;
   private int xval, yval, zval;
   public Set myHash;

   // Constructor
   public Cube100m(int Xval, int Yval, int Zval)
   {
      this.Xval = Xval;
      this.Yval = Yval;
      this.Zval = Zval;

      // Use div and mod to round x, y, and z
      // to the nearest 100; this is all we
      // need to calculate hashcode for a given
      // 100m cube
      //
      if(Xval < 0)
         xval = (Xval/100*100 - 100);
      else
         xval = (Xval/100*100 + 100);
      if(Yval < 0)
         yval = (Yval/100*100 - 100);
      else
         yval = (Yval/100*100 + 100);

      zval = (Zval/100*100 + 100);
      myHash = new HashSet();
   }
```

```
// Default Constructor
public Cube100m()
{
   this.Xval = 0;
   this.Yval = 0;
   this.Zval = 0;
   myHash = new HashSet();
}


public int getXval()
{
   return this.xval;
}

public int getYval()
{
   return this.yval;
}

public int getZval()
{
   return this.zval;
}

public int getX()
{
   return this.Xval;
}

public int getY()
{
   return this.Yval;
}

public int getZ()
{
   return this.Zval;
}

// Overrides hashCode() method for class Object
public int hashCode()
{
   return (xval/100*7 + yval/100*13 + zval/100*19);
}


// Calculate hashcode for given x, y, z values; this allows a
// hashcode to be calculated without having to actually create
// a Cube100m object
//
public static int calcKey(int X, int Y, int Z)
{
   int tempX, tempY, tempZ;
```

```
        if(X < 0)
            tempX = (X/100*100 - 100);
        else
            tempX = (X/100*100 + 100);
        if(Y < 0)
            tempY = (Y/100*100 - 100);
        else
            tempY = (Y/100*100 + 100);

        tempZ = (Z/100*100 + 100);

        return(tempX/100*7 + tempY/100*13 + tempZ/100*19);
    }


    void printVal()
    {
        System.out.println("   100m Cube at "+this.xval+" "+this.yval+"
            "+this.zval);
    }


    // Overrides equals() method for class Object
    public boolean equals(Object aName)
    {
        if(!this.getClass().getName().equals(aName.getClass().getName()))
            return false;
        else
            return (this.hashCode()==((Cube100m)aName).hashCode());
    }

} // end Cube100m
```

Appendix N: MyPoints Source Code

```
/*
*     MyPoints
*
*     Developed by
*     Lt Michael W. Darwin, USAF
*     Last Modified: 29 Aug 99
*
*     Creates an object to hold the coordinates for an
*     individual data point.  Instances of this class
*     are inserted into Cube100m object.
*
*/

import java.util.*;
import javax.vecmath.Point3d;


public class MyPoints
{
   private int Xval;
   private int Yval;
   private int Zval;


  // Constructor
   public MyPoints(int Xval, int Yval, int Zval)
   {
      this.Xval = Xval;
      this.Yval = Yval;
      this.Zval = Zval;
   }


  // Constructor
   public MyPoints(int[] pointArray)
   {
      this.Xval = pointArray[0];
      this.Yval = pointArray[1];
      this.Zval = pointArray[2];
   }


   public int getXval()
   {
      return Xval;
   }

   public int getYval()
   {
      return Yval;
   }
```

```
    public int getZval()
    {
       return Zval;
    }


    public int hashCode()
    {
       return Xval*13+Yval*7+Zval;
    }


    void printVal()
    {
       System.out.println("Xval "+Xval+"  Yval "+Yval+"  Zval "+Zval);
    }


    public boolean equals(Object aName)
    {
       if(!this.getClass().getName().equals(aName.getClass().getName()))
          return false;
       else
          return ((Xval==((MyPoints)aName).getXval())&&
             (Yval==((MyPoints)aName).getYval())
             &&(Zval==((MyPoints)aName).getZval()));
    }


} // end MyPoints
```

Bibliography

Andrews, K., J. Wolte, and M. Pichler, 1997: Information Pyramids: A New Approach to Visualising Large Hierarchies. *Proceedings of Visualization Conference '97*, IEEE Computer Society Press, pp. 49-52.

Andrews, K. and H. Heidegger, 1998: Information Slices: Visualising and Exploring Large Hierarchies using Cascading, Semi-Circular Discs. IEEE Symposium on Information Visualization, Research Triangle Park, NC, October 1998.

Botts, M., 1995: Metrics and Benchmarks for Visualization. Panel discussion in *Proceedings of Visualization Conference '95*, IEEE Computer Society Press, pp. 422-423.

Brown, K. and D. Petersen, 1999: *Ready-to-Run Java 3D*, Wiley Computer Publishing, 400 pp.

Card, S.K., J. Mackinlay, and B. Schneiderman, 1999: *Readings in Information Visualization: Using Vision to Think*, Morgan Kaufmann Publishers, 712 pp.

DeFanti, T.A., M.D. Brown, and B.H. McCormick, 1999: Visualization—Expanding Scientific and Engineering Research Opportunities, in *Readings in Information Visualization: Using Vision to Think*. Ed. Card, S.K., J. Mackinlay, and B. Schneiderman. Morgan Kaufmann Publishers, 712 pp.

Eckel, B., 1998: *Thinking in Java*, Prentice-Hall, 1098 pp.

Elvins, T. and B. Hibbard, 1998: VisAD: Connecting People to Computations and People to People. *SIGGRAPH Computer Graphics Newsletter*, Vol. 32 No. 3.

Gallagher, R.S., 1995: Scientific Visualization: An Engineering Perspective. CRC Press, 312 pp.

Global Hydrology Resource Center web site, as of 01 February 2000: http://thunder.msfc.nasa.gov/data/

Harold, E.R., 1999: *Java I/O*, O'Reilly and Associates, 568 pp.

Horstmann, C.S. and G. Cornell, 1999: *Core Java Volume I—Fundamentals*, Sun Microsystems Press, 742 pp.

Horton, I., 1997: *Beginning Java*, Wrox Press, 1039 pp.

Jerding, D. and J. Stasko, 1996: The Information Mural: Increasing Information Bandwidth in Visualizations. *Technical Report GIT-GVU-96-25*, Georgia Institute of Technology, October 1996.

Keller, P.R. and M.M. Keller, 1993: *Practical Data Visualization*. IEEE Computer Society Press, 229 pp.

Krider, E.P., R.C. Noggle, A.E. Pifer, and D.L. Vance, 1980: Lightning Direction-Finding Systems for Forest Fire Detection. *Bulletin American Meteorological Society*, Vol. 61, No. 9, pp. 980-986.

Lennon, C.L., 1979: Cloud-to-Ground Lightning Detector. In *NASA Tech Briefs*, Spring 1979, Vol. 4, No. 1.

Lucas, B., G.D. Abram, N.S. Collins, D.A. Epstein, D.L. Gresh, and K.P. McAuliffe, 1992: An Architecture for a Scientific Visualization System. *Proceedings of Visualization Conference '92*, IEEE Computer Society Press, pp. 107-113.

MacGorman, D.R. and W.D. Rust, 1998: *The Electrical Nature of Storms*, Oxford University Press, 422 pp.

Maier, L., C. Lennon, T. Britt, and S. Schaefer, 1995: Lightning Detection and Ranging (LDAR) System Performance Analysis. *6th Conference on Aviation Weather Systems*, American Meteorological Society Press, pp. 305-309.

Michaels, C. and M. Bailey, 1997: VisWiz: A Java Applet for Interactive 3D Scientific Visualization on the Web. *Proceedings of Visualization Conference '97*, IEEE Computer Society Press, pp. 261-267.

Pang, A. and N. Alper, 1994: Mix&Match: A Construction Kit for Visualization. *Proceedings of Visualization Conference '94*, IEEE Computer Society Press, pp. 302-306.

Poehler, H.A. and C.L. Lennon, 1979: Lightning Detection and Ranging System (LDAR) System Description & Performance Objectives. NASA Technical Memorandum 74105, 86 pp.

Roeder, W.P., 1999: Chief of Operations Support Flight and Science and Technical and Training Officer for the 45th Weather Squadron, PAFB, FL. Personal communication over the course of thesis preparation.

Schneiderman, B., 1996: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. *Proceedings of 1996 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, pp. 336-343.

Schroeder, W.J., K.M. Martin, and W.E. Lorensen, 1996: The Design and Implementation Of An Object-Oriented Toolkit For 3D Graphics And Visualization. *Proceedings of Visualization Conference '96*, IEEE Computer Society Press, pp. 93-100.

Sowizral, H.A. and M.F. Deering, 1999: The Java 3D API and Virtual Reality. *IEEE Computer Graphics and Applications*, May/June '99, IEEE Computer Society Press, pp. 12-15.

Star, J.L., 1993: Is Visualization Really Necessary? The Role of Visualization in Science, Engineering, and Medicine. Panel discussion in *Proceedings of Visualization Conference '93*, IEEE Computer Society Press, pp. 343-346.

Treinish, L.A., 1998: Task-Specific Visualization Design: A Case Study in Operational Forecasting. *Proceedings of Visualization Conference '98*, IEEE Computer Society Press, pp. 405-408.

Upson, C., T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, 1989: The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, July 1989, IEEE Computer Society Press, pp. 30-41.

Wood, J., K. Brodlie, and H. Wright, 1996: Visualization Over The World Wide Web And Its Application To Environmental Data. *Proceedings of Visualization Conference '96*, IEEE Computer Society Press, pp. 81-86.

Xiao, Y., J.P. Ziebarth, C. Woodbury, E. Bayer, B. Rundell, and J. van der Zijp, 1996: The Challenges of Visualizing and Modeling Environmental Data. *Proceedings of Visualization Conference '96*, IEEE Computer Society Press, pp. 413-415.

Vita

Michael W. (Mickey) Darwin was born on August 27, 1962 in Fort Worth, Texas, where his father was a firefighter for the Fort Worth Fire Department. Mickey graduated from Burleson High School in 1980 and enlisted in the Air Force in 1984. His first assignment was at Peterson AFB, Colorado where he was a Satellite Communications Technician. Assignments to Alzey Air Station, Germany, and Falcon AFB, Colorado followed in 1987 and 1990, respectively. In 1991 he was accepted into the Airman Education and Commissioning Program and was sent to the University of Oklahoma to receive his bachelors degree in meteorology. Following his first tour as an officer at Grand Forks AFB, North Dakota, he was accepted by the Air Force Institute of Technology as a masters degree student in meteorology. After completing his masters degree he will be reassigned to Offutt AFB, Nebraska to work at the Air Force Weather Agency.

Mickey was married to his wife, the former Shawna Lynn Northamer of West Branch, Iowa, in 1987 at the Hahn AFB Chapel in Germany. He has two children: Caitlin, 8 years old, and Evan, 5 years old.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

4. TITLE AND SUBTITLE
A 3D DISPLAY SYSTEM FOR LIGHTNING DETECTION AND RANGING (LDAR) DATA

5. FUNDING NUMBERS

6. AUTHOR(S)
Michael W. Darwin, 1st Lt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P. Street, Building 640
WPAFB OH 45433-7765 DSN: 785-4539

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GM/ENP/00M-05

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
45 WS/SYR
Attn: Mr. William P. Roeder
1201 Edward H. White II St., MS 7302
Patrick AFB, FL 32925-3238 DSN: 467-8410

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Lt Col Cecilia A. Miner, ENP, DSN: 785-3636, ext. 4645

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Lightning detection is an essential part of safety and resource protection at Cape Canaveral. In order to meet the unique needs of launching space vehicles in the thunderstorm prone Florida environment, Cape Canaveral has the only operational three-dimensional (3D) lightning detection network in the world, the Lightning Detection and Ranging (LDAR) system. Although lightning activity is detected in three dimensions, the current LDAR display, developed 20 years ago, is two-dimensional. This thesis uses modern three-dimensional graphics, object-oriented software design, and an innovative new visualization technique, called Nested Cubes, to develop a 3D visualization application for LDAR data.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Lightning, Lightning Detection and Ranging (LDAR), Visualization	109
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL